CRIMSON 2
USER MANUAL

red lion

Copyright © 2003 Red Lion Controls LP.

All Rights Reserved Worldwide.

The information contained herein is provided in good faith, but is subject to change without notice. It is supplied with no warranty whatsoever, and does not represent a commitment on the part of Red Lion Controls. Companies, names and data used as examples herein are fictitious unless otherwise stated. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without the express written permission of Red Lion Controls

All trademarks are acknowledged as the property of their respective owners.

Written by Mike Granby, Jesse Benefiel and John Bonner.

TABLE OF CONTENTS

GETTING STARTED 1
SYSTEM REQUIREMENTS 1
INSTALLING THE SOFTWARE 1
INSTALLING THE USB DRIVERS 1
CRIMSON BASICS 3
MAIN SCREEN ICONS 3
COMMUNICATIONS 3
DATA TAGS 3
USER INTERFACE 4
PROGRAMMING 4
DATA LOGGER 4
WEB SERVER 4
MISCELLANEOUS 4
USING BALLOON HELP 5
WORKING WITH DATABASES 5
DOWNLOADING TO A TERMINAL 5
CONFIGURING THE LINK 6
SENDING THE DATABASE 6
EXTRACTING DATABASES 7
MOUNTING THE COMPACTFLASH 7
SENDING THE TIME 8
UPDATING VIA COMPACTFLASH 8
GURU MEDITATION CODES 9
CONFIGURING COMMUNICATIONS 11
SERIAL PORT USAGE 11
SELECTING A PROTOCOL 12
PROTOCOL OPTIONS 12
WORKING WITH DEVICES 13
ETHERNET CONFIGURATION 13
IP PARAMETERS 14
PHYSICAL LAYER 14
PROTOCOL SELECTION 14
SLAVE PROTOCOLS 15
SELECTING THE PROTOCOL 15
ADDING GATEWAY BLOCKS 16
ADDING ITEMS TO A BLOCK 16
ACCESSING INDIVIDUAL BITS 17
PROTOCOL CONVERSION 17
MASTER AND SLAVE 17
MASTER AND MASTER 18
WHICH WAY AROUND? 18
DATA TRANSFORMATION 18
NOTES FOR EDICT USERS 19

CONFIGURING DATA TAGS 21
ALL ABOUT TAGS 21
TYPES OF TAGS 21
WHY USE TAGS? 22
CREATING TAGS 23
EDITING TAGS 23
EDITING PROPERTIES 24
EXPRESSION PROPERTIES 24
TRANSLATABLE STRINGS 25
EDITING FLAG TAGS 25
THE DATA TAB (VARIABLES) 26
THE DATA TAB (FORMULAE) 27
THE DATA TAB (ARRAYS) 27
THE FORMAT TAB 28
THE ALARMS TAB 28
THE TRIGGERS TAB 30
EDITING INTEGER TAGS 30
THE DATA TAB (VARIABLES) 30
THE DATA TAB (FORMULAE) 32
THE DATA TAB (ARRAYS) 32
THE FORMAT TAB 33
THE ALARM TABS 34
THE TRIGGERS TAB 35
EDITING MULTI TAGS 35
THE DATA TAB (VARIABLES) 36
THE DATA TAB (FORMULAE) 36
THE DATA TAB (ARRAYS) 37
THE FORMAT TAB 37
THE ALARM TABS 37
THE TRIGGERS TAB 38
EDITING REAL TAGS 39
EDITING STRING TAGS 39
THE DATA TAB (VARIABLES) 39
THE DATA TAB (FORMULAE) 39
THE DATA TAB (ARRAYS) 40
THE FORMAT TAB 40
MORE THAN TWO ALARMS 41
NOTES FOR EDICT USERS 42
CONFIGURING THE USER INTERFACE 43
CONTROLLING THE VIEW 43
OTHER VIEW OPTIONS 43
USING THE PAGE LIST 44
DISPLAY EDITOR TOOLBOXES 44
THE DRAWING TOOLBOX 44
THE FILL FORMAT TOOLBOX 44
THE LINE FORMAT TOOLBOX 44
THE TEXT FORMAT TOOLBOX 44

THE FOREGROUND TOOLBOX 45
THE BACKGROUND TOOLBOX 45
ADDING DISPLAY PRIMITIVES 45
SMART ALIGNMENT 45
KEYBOARD OPTIONS 46
LOCK INSERT MODE 46
SELECTING PRIMITIVES 46
MOVING AND RESIZING 47
REORDERING PRIMITIVES 47
EDITING PRIMITIVES 48
PRIMITIVE DESCRIPTIONS 48
THE LINE PRIMITIVE 48
THE SIMPLE GEOMETRIC PRIMITIVES 48
THE TANK PRIMITIVES 49
THE SIMPLE BAR-GRAPH PRIMITIVES 49
THE FIXED TEXT PRIMITIVE 50
THE AUTO TAG PRIMITIVE 51
THE TAG TEXT PRIMITIVES 51
EDITING THE UNDERLYING TAG 54
THE TIME AND DATE PRIMITIVE 54
THE RICH BAR-GRAPH PRIMITIVES 56
THE SYSTEM PRIMITIVES 57
DEFINING PAGE PROPERTIES 58
DEFINING SYSTEM ACTIONS 59
DEFINING KEY BEHAVIOR 59
ENABLING ACTIONS 60
ACTION DESCRIPTIONS 60
THE GOTO PAGE ACTION 60
THE PUSH BUTTON ACTION 61
THE CHANGE INTEGER VALUE ACTION 61
THE RAMP INTEGER VALUE ACTION 62
THE PLAY DIXIE ACTION 62
THE USER DEFINED ACTION 63
BLOCK DEFAULT ACTION 63
CHANGING THE LANGUAGE 64
ADVANCED TOPICS 64
ACTION PROCESSING 64
DATA AVAILABILITY 65
NOTES FOR EDICT USERS 65
CONFIGURING PROGRAMS 67
USING THE PROGRAM LIST 67
EDITING PROGRAMS 67
PROGRAM PROPERTIES 67
ADDING COMMENTS 69
RETURNING VALUES 69
HERE BE DRAGONS! 69

PROGRAMMING TIPS 70
MULTIPLE ACTIONS 70
IF STATEMENTS 70
SWITCH STATEMENTS 71
LOCAL VARIABLES 72
LOOP CONSTRUCTS 72
NOTES FOR EDICT USERS 74
CONFIGURING DATA LOGGING 75
CREATING DATA LOGS 75
USING THE LOG LIST 75
DATA LOG PROPERTIES 76
LOG FILE STORAGE 76
THE LOGGING PROCESS 77
ACCESSING LOG FILES 78
USING WEBSYNC 78
WEBSYNC SYNTAX 78
OPTIONAL SWITCHES 78
EXAMPLE USAGE 79
NOTES FOR EDICT USERS 79
CONFIGURING THE WEB SERVER 81
WEB SERVER PROPERTIES 81
ADDING WEB PAGES 82
USING A CUSTOM WEB SITE 83
CREATING THE SITE 83
EMBEDDING DATA 83
DEPLOYING THE SITE 83
COMPACTFLASH ACCESS 83
WEB SERVER SAMPLES 84
WRITING EXPRESSIONS 89
DATA VALUES 89
CONSTANTS 89
TAG VALUES 90
COMMUNICATIONS REFERENCES 91
SIMPLE MATH 91
OPERATOR PRIORITY 91
TYPE CONVERSION 91
COMPARING VALUES 92
TESTING BITS 92
MULTIPLE CONDITIONS 93
CHOOSING VALUES 93
MANIPULATING BITS 94
AND, OR AND XOR 94
SHIFT OPERATORS 94
BITWISE NOT 94

INDEXING ARRAYS 94
INDEXING STRINGS 95
ADDING STRINGS 95
CALLING PROGRAMS 95
USING FUNCTIONS 95
PRIORITY SUMMARY 95
NOTES FOR EDICT USERS 96
WRITING ACTIONS 97
CHANGING PAGE 97
CHANGING NUMERIC VALUES 97
SIMPLE ASSIGNMENT 97
COMPOUND ASSIGNMENT 97
INCREMENT AND DECREMENT 97
CHANGING BIT VALUES 97
RUNNING PROGRAMS 98
USING FUNCTIONS 98
OPERATOR PRIORITY 98
NOTES FOR EDICT USERS 98
USING RAW PORTS 99
CONFIGURING A SERIAL PORT 99
CONFIGURING A TCP/IP SOCKET 99
READING CHARACTERS 100
READING ENTIRE FRAMES 100
SENDING DATA 101
NOTES FOR EDICT USERS 101
FUNCTION REFERENCE 103
NOTES FOR EDICT USERS 103
ABS(*VALUE*) 104
BEEP(*FREQ*, *PERIOD*) 105
COPY(*DEST*, *SRC*, *COUNT*) 106
COS(*THETA*) 107
DATE(*Y, M, D*) 108
DEG2RAD(*THETA*) 109
FILL(ELEMENT, DATA, COUNT) 110
GETDATE (*TIME*) AND FAMILY 111
GETMONTHDAYS(*Y, M*) 112
GETNOW() 113
GOTOPAGE(*NAME*) 114
INTTOTEXT(*DATA*, *RADIX*, *COUNT*) 115
LEFT(*STRING*, *COUNT*) 116
LEN(*STRING*) 117
MAKEINT(*VALUE*) 118

MAKEFLOAT(*VALUE*)....119
MAX(*A, B*)....120
MEAN(*ELEMENT, COUNT*)....121
MID(*STRING, POS, COUNT*)....122
MIN(*A, B*)....123
MUTESIREN() 124
NOP()....125
PI() 126
PLAYRTTTL (*TUNE*) 127
POPDEV(*ELEMENT, COUNT*) 128
PORTINPUT(*PORT, START, END, TIMEOUT, LENGTH*)....129
PORTPRINT(*PORT, STRING*) 130
PORTREAD(*PORT, PERIOD*)....131
PORTWRITE(*PORT, DATA*)....132
POWER(*VALUE, POWER*)....133
RAD2DEG(*THETA*)....134
RANDOM(*RANGE*) 135
RIGHT(*STRING, COUNT*)....136
SETLANGUAGE(*CODE*)....137
SGN(*VALUE*) 138
SIN(*THETA*) 139
SIRENON()....140
SLEEP(*PERIOD*)....141
SQRT(*VALUE*)....142
STDDEV(*ELEMENT, COUNT*) 143
SUM(*ELEMENT, COUNT*) 144
TAN(*THETA*)....145
TEXTTOINT(*STRING, RADIX*) 146
TIME(*H, M, S*)....147

CRIMSON 2
USER MANUAL

GETTING STARTED

Welcome to Crimson 2—the latest operator interface configuration package from Red Lion Controls. Crimson is designed to provide quick and easy access to the features of the G3 series of operator panels, while still allowing the advanced user to take advantage of high-end features, such as Crimson's unique programming support.

SYSTEM REQUIREMENTS

Crimson 2 is designed to run on PCs with the following specifications…

- A Pentium class processor as required by the chosen operating system.
- RAM and free disk space as required by the chosen operating system.
- An additional 10MB of disk space for software installation.
- A display of at least 800 by 600 pixels, with 256 or more colors recommended.
- An RS-232 or USB port for downloading to a G3 panel.

Crimson 2 is designed to operate with all versions of Microsoft Windows from Windows 95 upwards. If you want to take advantage of the USB port provided by the G3 operator panels, you will need to use, as a minimum, Windows 98. If you intend to use the USB port to remotely access the G3's CompactFlash card, we recommend that you use Windows 2000 or Windows XP. While Windows 98 is capable of accessing the card, the later versions of the operating system provide more robust operation, and are much better about when they choose to lock the card, thereby preventing the C2 runtime from writing data.

INSTALLING THE SOFTWARE

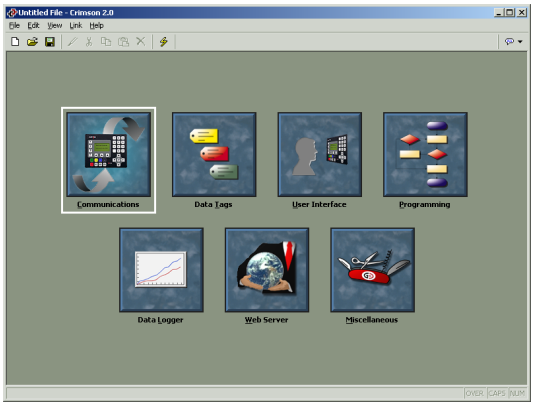
If you downloaded the Crimson software from Red Lion's website, simply execute the download file, and follow the instructions. If you received a copy of Crimson on CD, place the CD in your system's CDROM drive, and follow the instructions that will appear. If no instructions appear, you may have auto-run disabled. In that case, select the Run option from the Start menu, and enter **`x:\setup`**, where **`x`** is the drive letter of your CDROM drive. Again, follow the resulting instructions, and the software will be installed.

INSTALLING THE USB DRIVERS

When you first connect a G3 panel to your PC using a USB cable, Windows will prompt you for the location of the drivers for the device. The default location for these drivers is C:\Program Files\Red Lion Controls\Crimson 2.0\Device. When the Hardware Setup Wizard appears, choose the Browse option, and either point the Wizard at that location or whatever other location you specified during installation of the software. It is important that you perform this step correctly, or you may have to manually remove the drivers using the Device Manager, and repeat the installation once more.

CRIMSON BASICS

To run Crimson, select the Crimson icon from the Red Lion Controls folder on the Programs section of your Start Menu. The main C2 screen will appear, showing the icons that are used to configure the various aspects of the operator panel's behavior…



The software is designed such that the first three icons are the only ones required for the majority of simple applications. The remainder of the icons provide access to the terminal's more advanced features, such as programming, data logging and the G3's web server.

MAIN SCREEN ICONS

The sections below provide an overview of each icon in turn…

COMMUNICATIONS

This icon is used to specify which protocols are to be used on the G3's serial ports and on the Ethernet port. Where master protocols are used (ie. protocols by which the G3 initiates data transfer to and from a remote device) you can also use this icon to specify one or more devices to be accessed. Where slave protocols are used (ie. protocols by which the G3 receives and responds to requests from remote devices or computer systems) you can specify which data items are to be exposed for read or write access. You can also use this icon to move data between one remote device and another via Crimson's protocol converter.

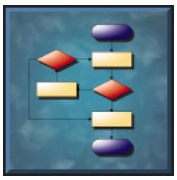
DATA TAGS

This icon is used to define the data items to be accessed within the remote devices, or to define internal data items to store information within the terminal itself. Each tag has a variety of properties associated with it. The most basic property is formatting data, which is used to specify how the data held within a tag is to be shown on the terminal's display, and on such things as web pages. By specifying this information within the tag, Crimson removes the need for you to re-enter formatting data each time a tag is displayed. More advanced tag properties include alarms that may activate when various conditions relating to the tag occur, or triggers, which perform programmable actions on similar conditions.

USER INTERFACE

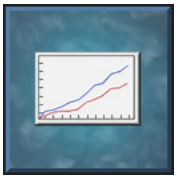
This icon is used to create and edit display pages, and to specify what actions should be taken when the operator panel's keys are pressed, released or held down. The page editor allows you to display various graphical items known as primitives. These vary from simple items, such as rectangles and lines, to more complex items that can be tied to the value of a particular tag or expression. By default, such primitives use the formatting information defined when the tag was created, but this information can be overridden if required.

PROGRAMMING



This icon is used to create and edit programs using C2's unique C-like programming language. These programs can perform complex decision making or data manipulation operations based upon any data items within the system. They serve to extend the functionality of Crimson beyond that of the standard functions included in the software, thereby ensuring that even the most complex applications can be tackled with ease.

DATA LOGGER



This icon is used to create and manage data logs, each of which can record any number of variables to the G3's CompactFlash card. Data may be recorded as quickly as once per second. The recorded values will be stored in CSV (comma separated variable) files that can easily be imported into applications such as Microsoft Excel. The files can be accessed by swapping-out the CompactFlash card, by mounting the card as a drive on a PC connected on the G3's USB port, or by accessing them via Crimson's web server via the Ethernet port.

WEB SERVER

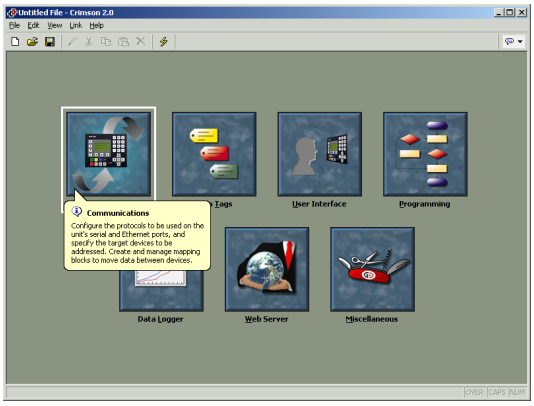
This icon is used to configure Crimson's web server and to create and edit web pages. The web server is capable of providing remote access to the G3 via a number of mechanisms. First, you can use Crimson to create automatic web pages which contain lists of tags, each formatted according to the tag's properties. Second, you can create a custom site using a third party HTML editor such as Microsoft FrontPage, and then include special text to instruct Crimson to insert live tag values. Finally, you can enable C2's unique remote access and control feature, which allows a web browser to view the G3's display and control its keyboard. The web server can also be used to access CSV files from the Data Logger.

MISCELLANEOUS

This icon is reserved for future expansion.

USING BALLOON HELP

Crimson provides a useful feature called Balloon Help...


Untitled File - Crimson 2.0
File Edit View Link Help
Communications
Configure the protocols to be used on the unit's serial and Ethernet ports, and specify the target devices to be addressed. Create and manage mapping blocks to move data between devices.
Tags
User Interface
Programming
Data Logger
Web Server
Miscellaneous

This feature allows you to see help information for each icon in the main menu, or for each field in a dialog box or window. It is controlled via the icon at the right-hand edge of the toolbar, and can be configured to three modes, namely "Do Not Display", in which case balloon help is disabled; "When Mouse Over", in which case help is displayed when the mouse pointer is held over a particular field for a certain period of time; or "When Selected", in which case help is always displayed for the currently selected field.

WORKING WITH DATABASES

Crimson stores all the information about a particular panel's configuration in what is called a database file. These files have the extension of `CD2`, although Windows Explorer will hide this extension if it is left in its default configuration. Crimson database files differ from those used by previous Red Lion operator panels, in that they are text files which are thus far easier to recover in the case of accidental corruption. Databases are manipulated via the commands found on the File menu. These commands are standard for all Windows applications, and need no further explanation. The exception is Save Image, which will be covered later.

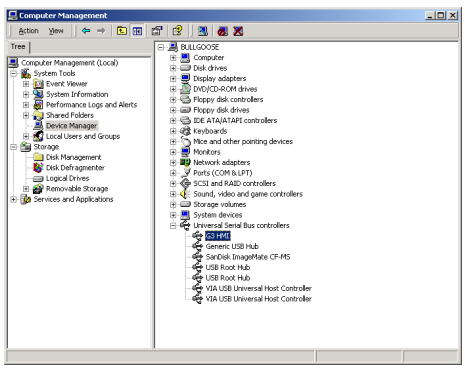
DOWNLOADING TO A TERMINAL

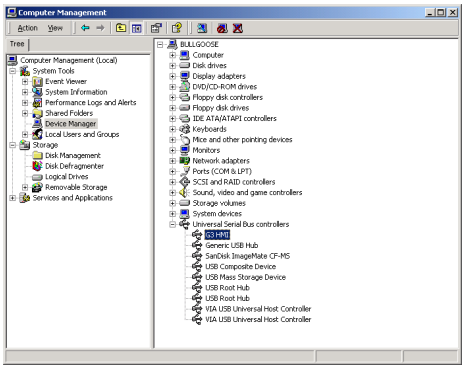
Crimson database files are downloaded to the G3 panel by means of the Link menu. The download process typically takes only a few seconds, but can take somewhat longer on the first download if Crimson has to update the firmware in the operator panel, or if the panel does not contain an older version of the current database. After this first download, however, Crimson uses a process known as incremental download to ensure that only changes to the database are transferred. This means that changes can be made in seconds, thereby reducing your development cycle time and simplifying the debugging process.

CONFIGURING THE LINK

The programming link between the PC and the G3 is made using either an RS-232 serial port, or a USB port. Before downloading, you should use the Link-Options command to ensure that you have the correct serial port selected, or that you have selected USB as appropriate.

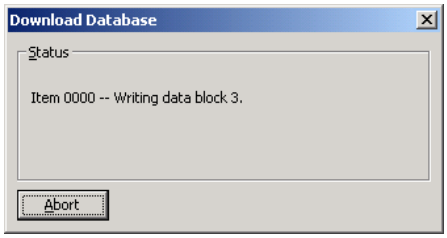
If you are using USB, you might also want to ensure that the G3's USB drivers have been correctly installed. To do this, connect the G3 panel, and, if the drivers have not previously been installed, follow the instructions at the start of this manual. Then, open the Device Manager for your operating system, and expand the USB icon to show the icon for the G3 Panel device. Ensure that this icon does not display a warning symbol. If it does, remove the device, unplug and reconnect the G3 panel, and verify that you have correctly followed the driver installation procedure. The illustrations below show typical Device Manager views with the CompactFlash dismounted and mounted, respectively….





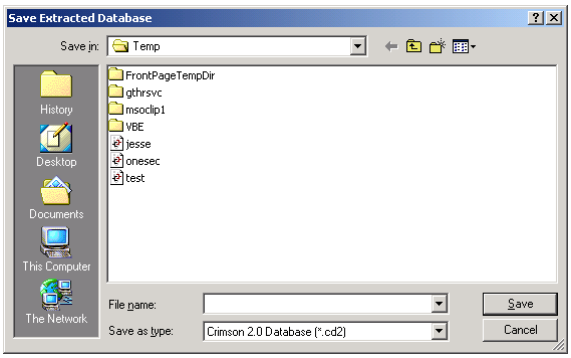
SENDING THE DATABASE

Once the link is configured, the database can be downloaded using either the Link-Send or Link-Update commands. The former will send the entire database, whether or not individual objects within the file have changed. The latter will only send changes, and will typically take a much shorter period of time to complete. The Update command is typically the only one that you will need, as Crimson will automatically fall-back to a complete send if the incremental download fails for any reason. As a shortcut, note that you can access Link Update via the lightning-bolt symbol on the toolbar, or via the **F9** key on the PC.



EXTRACTING DATABASES

The Link-Support Upload command can be used to instruct Crimson whether or not it should include the information necessary to support database upload when sending a database to a G3 panel. Supporting upload will slow the download process somewhat, but will ensure that should you lose your database file, you will be able to extract an editable image from the terminal. If you lose your database file and you do not have upload support enabled, you will not be able to reconstruct your file without starting from scratch. To extract a database from a panel, use the Link-Extract command. This command will upload the database, and then prompt you for a name under which to save the file. The file will then be opened for editing.



MOUNTING THE COMPACTFLASH

If you are connected to a G3 panel via the USB port, you can instruct Crimson to mount the G3's CompactFlash card as a drive within Windows Explorer. You can use this functionality to save files to the card or to read information from the Data Logger. The drive is mounted and dismounted by sending commands using the Mount Flash and Dismount Flash options on the Link menu. Once a command has been sent, the G3 panel will be reset, and Windows will refresh the appropriate Explorer windows to show or hide the CompactFlash drive.

Note that some caution is required when mounting the CompactFlash card...

- When the card is mounted, the G3 will periodically inform the PC if data on the card has been modified. This means that both the PC and the G3 will suffer performance hits if the card is mounted during data logging operations for longer than necessary.

- If you write to the CompactFlash card from your PC, the G3 will not be able to access the card until Windows releases its "lock" on the card's contents. This may take up to a minute, and will restrict data logging operations during that time, and prevent access to custom web pages. Crimson will use the G3's RAM to ensure that no data is lost, but if too many writes are performed such that the card is kept locked for four minutes or more, data may discarded. Note that Windows 98 is particularly bad at keeping the card locked when there is no need for it. Windows 2000 or Windows XP is thus the operating system of choice when using this feature.

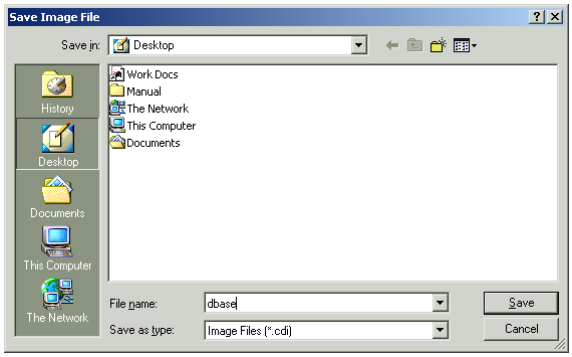
- You should never attempt to format a CompactFlash card that you have mounted via the G3, whether it be via Windows Explorer or from the command prompt. Windows does not correctly lock the card during format operations, and the format may thus be unreliable and lead to subsequent data loss. The approved method of formatting a card is via a stand-alone CompactFlash adapter.

SENDING THE TIME

The Link-Send Time command can be used to set the G3's clock to match that of the PC on which Crimson is executing. Obviously, make sure your clock is right before you do this!

UPDATING VIA COMPACTFLASH

If you need to update the database within a unit which is already installed at a customer's site, Crimson allows you to save a copy of the database to a CompactFlash card, ship that card to your customer, and have the G3 load the database from that card. The process is performed via the Save Image command on the File menu.



The Save Image command will create a Crimson database image file with a **CDI** extension. It will also save a copy of the current G3 firmware to a file with a **BIN** extension. The image file must be given the name **DBASE.CDI**, and both it and the **BIN** file must be placed in the root directory of a CompactFlash card. To update a G3 panel, power down the unit, insert the CompactFlash card bearing the two files, and reapply power to the unit. The G3's boot loader will first check whether it needs to upgrade the unit's firmware, and once this process has been completed, the Crimson runtime application will load the database stored on the card. The CompactFlash card can then be removed or left in place as required.

GURU MEDITATION CODES

If a problem with the Crimson runtime application within the G3 operator panel results in the panel being reset, the condition that caused the fault will be logged. When the panel restarts, this information will be displayed in the form of a Guru Meditation Code. A typical code will have the format…

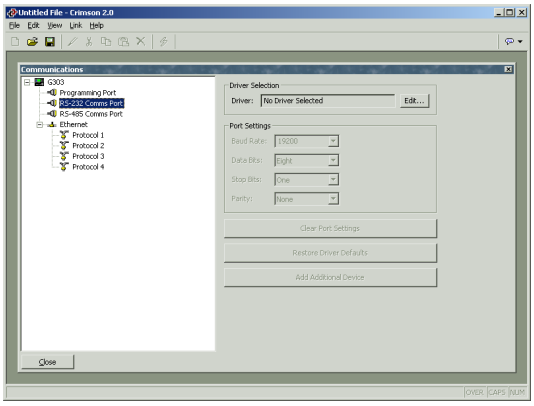
```
03-2004-1BE4-205
```

The message can be accepted by pressing the F1 key, at which point the terminal will resume normal operation. Note that communications, data logging and the web server are still active when the GMC is displayed—only the user interface is interrupted. This means that system disruption is minimized, and functions such as protocol conversion continue to operate.

Before accepting the message, you may wish to write down the code. You may then email it to Red Lion technical support, so that one of our technical gurus can meditate on this information in order to track-down the cause of the problem. You may also want to email a copy of the terminal's database, and describe what you were doing when the terminal crashed.

CONFIGURING COMMUNICATIONS

The first stage of creating a Crimson database is to configure the communications ports of the G3 panel to indicate which protocols you want to use, and which remote devices you want to access. These operations are performed from the Communications window, which is opened by selecting the first icon of the Crimson main screen.



As can be seen, the Communications window lists the unit's available ports in the form of a tree structure. G3 panels have three primary serial ports, with the option to add a further two ports in the form of an expansion card. They also provide a single Ethernet port which is capable of running four communications protocols simultaneously.

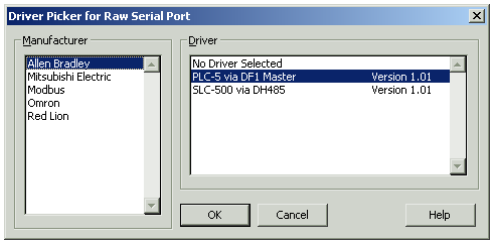
SERIAL PORT USAGE

When deciding which of the G3's serial ports to use for communications, note that…

- The G303 multiplexes a single serial communications controller between its RS-232 and RS-485 comms ports. This means that if either port is used for a slave protocol, the other port is unavailable. It also means that if a token-passing protocol such as Allen-Bradley DH-485 is employed, the other port is similarly disabled. Other G3 panels impose no such restrictions.
- The unit's programming port may be used as an additional communications port, but it will obviously not be available for download if it is so employed. This is not an issue if the USB port is used for such purposes, and it is highly recommended that you use this method of download if you want to connect serial devices via the programming port.

SELECTING A PROTOCOL

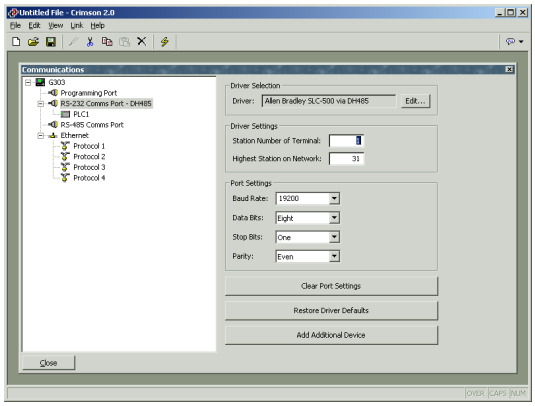
To select a protocol for a particular port, click on that port's icon in the left-hand pane of the Communications window, and press the Edit button next to the Driver field in the right-hand pane. The following dialog box will appear…



Select the appropriate manufacturer and driver, and press the OK button to close the dialog box. The port will then be configured to use the appropriate protocol, and a single device icon will be created in the left-hand pane. If you are configuring a serial port, the various Port Settings fields (Baud Rate, Data Bits, Stop Bits and Parity) will be set to values appropriate to the protocol in question. You should obviously check these settings to make sure that they correspond to the settings for the device to be addressed.

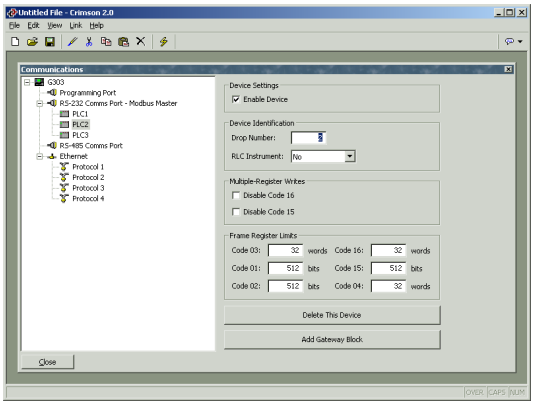
PROTOCOL OPTIONS

Some protocols require additional configuration of parameters specific to that protocol. These appear in the right-hand pane of the Communications window when the corresponding port icon is selected. The example below shows the additional parameters for the Allen-Bradley DH-485 driver, which appear under the Driver Settings section of the window.



WORKING WITH DEVICES

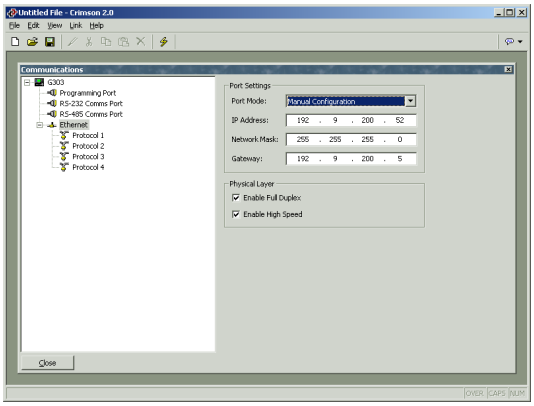
As mentioned above, when a communications protocol is selected, a single device is created under the corresponding port icon. In the case of a master protocol, this represents the initial remote device to be addressed via the protocol. If the protocol supports access to more than one device, you can use the Add Additional Device button included with the port icon's properties to add further target devices. Each device is represented via an icon in the left-hand pane of the Communications window, and, depending on the protocol in question, may have a number of properties to be configured…



In the example above, the Modbus Universal Master protocol has been selected, and two additional devices have been created, indicating that a total of three remote devices are to be accessed. The right-hand pane of the window shows the properties of a single device. The Enable Device property is present for devices for all protocols, while the balance of the fields are specific to the protocol that has been selected. Note that the devices are given default names by Crimson when they are created. These names may be changed by selecting the appropriate icon in the left-hand pane, and simply typing the new device name.

ETHERNET CONFIGURATION

The G3's Ethernet port is configured via the Ethernet icon in the left-hand pane of the Communications window. When this icon is selected, the following settings are displayed…



IP PARAMETERS

The Port Mode field controls whether or not the port is enabled, and the method by which the port is to obtain its IP configuration. If DHCP mode is selected, the G3 will attempt to obtain an IP address and associated parameters from a DHCP server on the local network. If the unit is configured to use slave protocols or to serve web pages, this option will only make sense if the DHCP server is configured to allocate a well-known IP address to the MAC address associated with the unit, as otherwise, users will not be sure how to address the panel!

If the more common Manual Configuration mode is selected, the IP Address, Network Mask and Gateway fields must be filled out with the appropriate information. The default values provided for these fields will almost never be suitable for your application! Be sure to consult your network administrator when selecting appropriate values, and be sure to enter and download these values before connecting the G3 to your network. If you do not follow this advice, it is possible—although unlikely—that you will cause problems on your network.

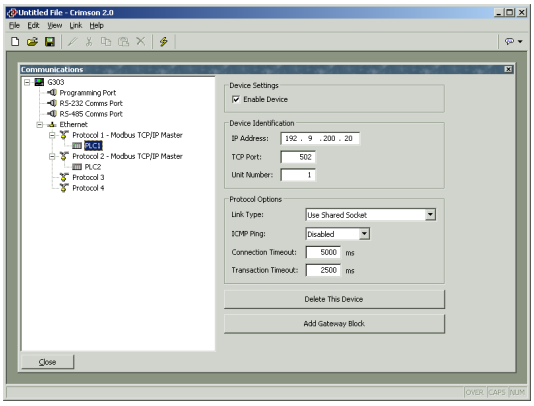
PHYSICAL LAYER

The Physical Layer options control the type of connection that the G3 will attempt to negotiate with the hub to which it is connected. Generally, these options can be left in their default states, but if you have trouble establishing a reliable connection, especially when connecting directly to a PC without an intervening hub or switch, consider turning off both Full Duplex and High Speed operation to see if this solves the problem.

PROTOCOL SELECTION

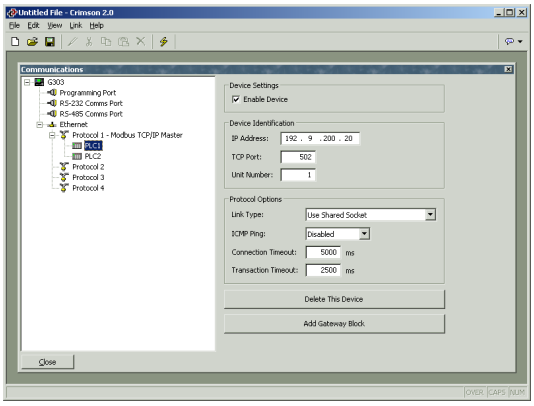
Once the Ethernet port has been configured, you can select the protocols that you wish to use for communications. Up to four protocols may be used at once, and many of these protocols will support multiple remote devices. This means that you have several options when deciding how to mix protocols and devices to achieve the results you want. For example, suppose you want to connect to two remote slave devices using Modbus over TCP/IP.

Your first option is to use two of the Ethernet port's protocols, and configure both as Modbus TCP/IP Masters, with a single device attached to each protocol...



For most protocols, this will produce higher performance, as it will allow simultaneous communications with the two devices. It will, however, consume two of the four available protocols, limiting your ability to connect via additional protocols in complex applications.

Your second option is to use a single protocol configured as a Modbus TCP/IP Master, but to add a further device so that both slaves are accessed via the same driver…



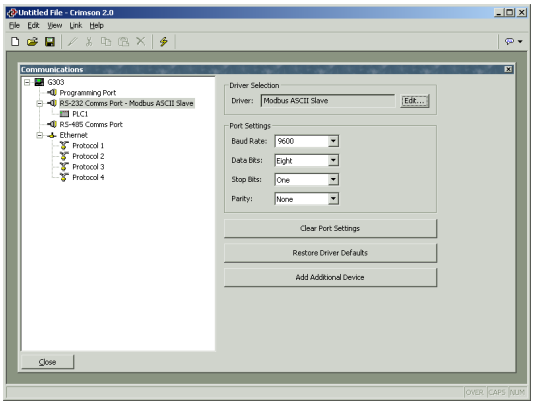
This will typically produce slightly reduced performance, as Crimson will poll each device in turn, rather than talking to both devices at the same time. It will, however, conserve Ethernet protocols, allowing more complex applications without running out of resources.

SLAVE PROTOCOLS

For master protocols (ie. those where the G3 initiates communication) there is no further configuration required under the Communications icon. For slave protocols (ie. those where the G3 receives and responds to remote requests), however, the process is slightly more complex, as you must also indicate what data you wish to expose for remote access.

SELECTING THE PROTOCOL

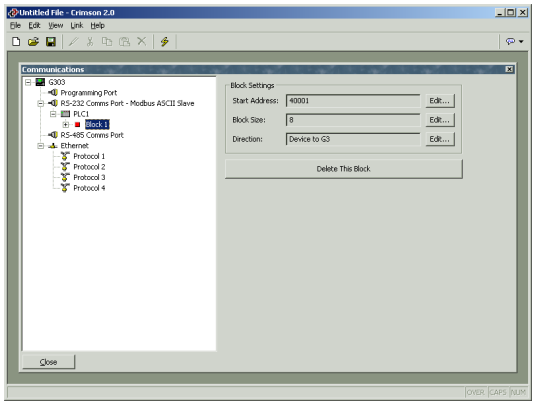
As with master protocols, the first stage is to select the protocol for the communications port that you wish to use. The example below shows the G3's RS-232 port configured for operation with the Modbus ASCII Slave protocol…



Note that a single device has been automatically created for the protocol. In the case of master protocols, this represents the remote device that the G3 will access. In this case, though, the device represents the Modbus slave that the G3 will itself embody. This means that only a single device is required, and that things such as the station number to which the G3 will respond are normally configured via the port settings rather than those of the device.

ADDING GATEWAY BLOCKS

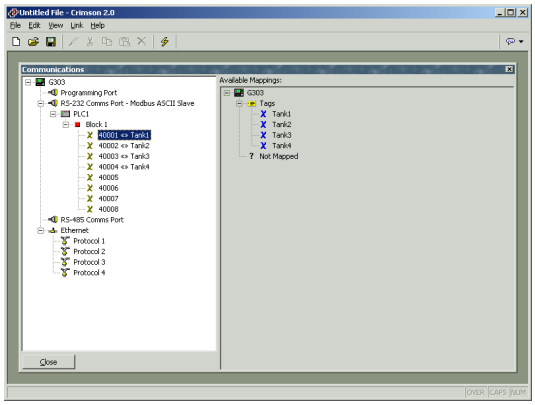
Having configured the protocol, you must now decide what range of addresses you want the slave protocol to expose. In this example, we want to use Modbus registers **40001** through **40008** to allow read and write access to certain data items in our database. We begin by selecting the device icon in the left-hand pane of the Communications window, and clicking the Add Gateway Block button in the right-hand pane. An icon to represent Block 1 will appear, and selecting it will show the following settings…



In the example above, we have configured the Start Address to **40001** to indicate that this is where we want the block to begin. We have also configured the Block Size to eight so as to allocate one Modbus register for each tag we want to expose. Finally, we have configured the Direction as Device to G3, to indicate that we want remote devices to be able to read and write data items exposed via this block.

ADDING ITEMS TO A BLOCK

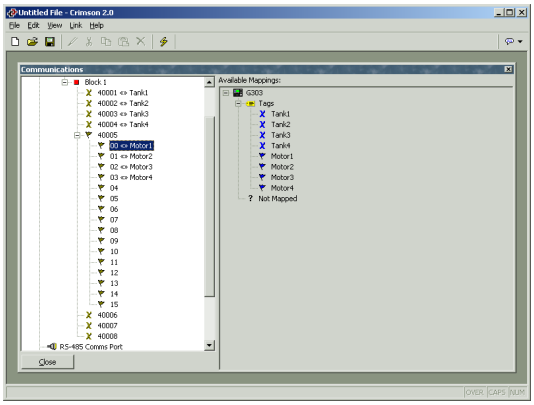
Once the block has been created and its size defined, entries appear in the left-hand pane of the window to represent each of the registers that the block exposes to remote access. When one of these entries is selected, the right-hand pane shows a list of available data items, comprising both tags from within your database, and data registers from any master communications devices that you have configured…



To indicate that you want a particular register within your gateway block to correspond to a particular data item, simply drag that item from the right-hand pane to the left-hand pane,

dropping it on the appropriate gateway block entry. The example above shows how the first four registers in the block have been mapped to tags called **Tank1** through **Tank4**, indicating that accesses to **40001** through **40004** should be mapped to the respective variables.

ACCESSING INDIVIDUAL BITS

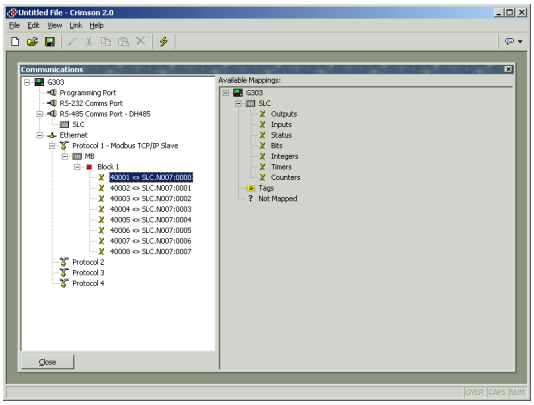


If your application requires it, you can expand individual elements within a Gateway Block to their constituent bits, and map a different data item to each bit. To do this, right-click on the element in question, and select Expand from the resulting pop-up menu. The right-hand pane will be updated to show the individual bits that make up the register, and these can be mapped using the drag-and-drop process described above.

PROTOCOL CONVERSION

In addition to exposing internal data tags via slave protocols, Gateway Blocks can also be used to expose data that is obtained from other remote devices, or to move data between two such master devices. This unique protocol conversion feature allows much tighter integration between elements of your control system, even when using simple, low-cost devices.

MASTER AND SLAVE



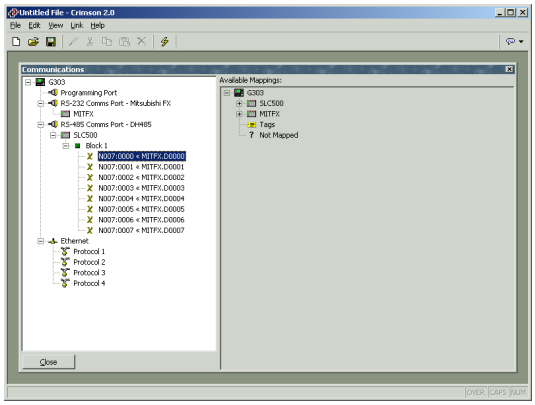
Exposing data from other devices over a slave protocol is simply an extension of the mapping process described above, except this time, instead of dragging a tag from the right-hand pane, you should expand the appropriate master device, and drag across the icon that represents the

registers that you want to expose. You will then be asked for a start address in the master device, and the number of registers to map, and the mappings will be created as shown.

In this example, registers **N7:0** through **N7:7** in an Allen-Bradley controller have been exposed for access via Modbus TCP/IP as registers **40001** through **40008**. Crimson will automatically ensure that these data items are read from the Allen-Bradley PLC so as to fulfill Modbus requests, and will automatically convert writes to the Modbus registers into writes to the PLC. This mechanism allows even simple PLCs to be connected on an Ethernet network.

MASTER AND MASTER

To move data between two master devices, simply select one of the devices, and create a Gateway Block for that device. You can then add references to the other device's registers just as you would when exposing data on a slave protocol. Again, C2 will automatically read or write the data as required, transparently moving data between the devices. The example below shows how to move data from a Mitsubishi FX into an Allen-Bradley PLC…



WHICH WAY AROUND?

One question that may occur to you is whether you should create the Gateway Block within the Allen-Bradley device, as in this example, or within the Mitsubishi device. The first thing to note is that there is no need to create more than a single block to perform transfers in a single direction. If you create a block in AB to read from MITFX, and a block in MITFX to write to AB, you'll simply perform the transfer twice and slow everything down! The second observation is that the decision as to which device should "own" the Gateway Block is essentially arbitrary. In general, you should create your blocks so as to minimize the number of blocks in the database. This means that if the registers in the Allen-Bradley lay within a single range, but the registers in the Mitsubishi are scattered all over the PLC, the Gateway Block should be created within the Allen-Bradley device so as to remove the need to create multiple blocks to access the different ranges of the Mitsubishi device.

DATA TRANSFORMATION

You may also use Gateway Blocks to perform math operations that your PLC might not otherwise be able to handle. For example, you may want to read a register from the PLC, scale it, take the square root, and write it back to another PLC register. To accomplish this, refer to the section on Data Tags, and create a mapped variable to represent the input value

that will be read from the device. Then, create a formula to represent the output value, setting the expression so as to perform the required math. You can then create a Gateway Block targeted at the required output register, and drag the formula across to instruct Crimson to write the derived value back to the PLC.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 operator interface configuration software will by this point no doubt be wondering what happened to the Communications Blocks that they know and love so well. The answer is simple: Crimson manages communications blocks automatically, and only reads and writes the data that is needed to satisfy the requirements of the system at that time. This means that if a register is only accessed on a given page, it will only be read when that page is selected. The communications process is thus automatically optimized.

Other communications differences between Edict-97 and Crimson are…

- Slave protocols are no longer handled via Communications Blocks, but by mapping data items into Gateway Blocks. This means that the same data item can be exposed via multiple slave protocols without any further configuration.
- Writes in Crimson are transaction-based rather than value-based. This means that if you write a register to 1 and then to 0, you are guaranteed that two writes will be performed. This avoids the need for Pulse Blocks and other horribleness.
- Rather than using the comms update complete event to move values from one device to another, or to transform values and write them back to the source device, Crimson uses Gateway Blocks as described above.
- Crimson's communications architecture operates at much higher performance levels than that of Edict-97. It uses multiple tasks and much greater amounts of buffering to ensure that communications updates are kept to a minimum.

CONFIGURING DATA TAGS

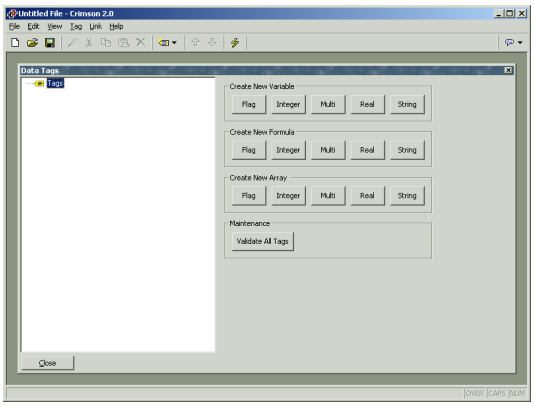
Once you have configured the communications options for your database, the next step is to define the data items that you want to display or otherwise manipulate. This is done by selecting the Data Tags icon from the main screen.

ALL ABOUT TAGS

Data Tags are named entities which represent data items within the operator terminal. Tags may be "mapped" to registers in remote devices, in which case Crimson will automatically read the corresponding register when the tag is referenced or displayed. Similarly, if you change a mapped tag, Crimson will automatically write the new value to the remote device.

TYPES OF TAGS

When you first open the Data Tags window, you will see that the right-hand pane contains an apparently bewildering number of buttons that can be used to create different kinds of data tags. While all these buttons may seem a little intimidating at first, the fifteen different kinds of tag can be broken down into three families, each containing five members.



TAG FAMILIES

The three families of tags are listed below…

- *Variables* represent a single data item within the terminal. Variables may be mapped to PLC registers, and may be configured as retentive, in which case their values will be kept in memory even when the operator panel is powered off. The defining characteristics of a variable are that they contain a single item, and that it is *in theory* possible to write to this item, even if in practice the variable is configured as read-only. (If this seems confusing, read on, and you'll see how this contrasts to a formula, which does not have this property.)
- *Formulae* represent derived values. They are a combination of other data items, typically combined using one or more math operations. For example, a formula might represent the sum of two tank levels. While a formula can be set to be equal to the contents of a PLC register, it is not truly mapped to that register, in that it can *never* be written to and thus cannot be considered to be equivalent to

that register. The need for this restriction is obvious if you consider a formula such a **`Tank1+Tank2`**. What would it mean to write to this expression?

- *Arrays* represent a collection of data items within the terminal. They cannot be mapped to PLC registers, but instead they represent a list of values within the panel's own memory. These are typically used to store recipe data, or to build up collections of data for statistical analysis. They are not used in the majority of simple applications, but provide a powerful tool for more complex projects.

TAG TYPES

Each family contains five tag types, each of which holds a different kind of data…

- *Flag* tags represent a single true or false condition. When they are mapped to a register in a remote device, they will typically correspond to an internal coil or to a single digital I/O point. Flag formulae typically represent combinations of such items, or comparisons of numeric values.

- *Integer* tags represent 32-bit signed numbers. These tags can store values between –2,147,483,648 and +2,147,483,647. Even if a tag is mapped to a PLC register which contains only 16 bits of data, Crimson performs its internal operations at the higher level of precision to ensure large intermediate values can be handled with ease.

- *Multi* tags represent numeric values that correspond to a number of distinct states. Thus, while an integer might represent a tank level, a multi tag will represent, say, one of three states of a machine, such as Stopped, Running or Paused. The distinction is obvious when you consider that a multi tag is displayed as one of a set of strings, while an integer is displayed as a number.

- *Real* tags represent 32-bit single precision floating-point numbers. These tags can store values between $\pm 10^{-38}$ and $\pm 10^{+38}$ with a precision of about 7 significant figures. While it is seldom necessary to use real tags to represent physical quantities—which typically have more tightly defined ranges—they are useful for performing statistical operations or other math functions.

- *String* tags represent an item of text made up of a number of characters. They are used to store such things as recipe names, or to process data received using Raw Port device drivers. Strings cannot be mapped to PLC registers, but can be used to store such data within the terminal itself.

WHY USE TAGS?

Given all these various options, you may wonder why you would want to use tags in the first place? After all, Crimson allows you to directly place a PLC register on a display page, so you can in fact configure a simple database without ever opening the Data Tags icon. The basic answers are as follows…

- Tags allow you to name data items, so you know which data item within the PLC you are referring to. Further, if the data in the PLC moves or if you decide to

switch to an entirely different family of PLC, you can simply re-map the tags, and avoid having to make any other changes to your database.

- Tags allow you to avoid re-entering the same information again and again. When you create a tag, you specify how the tag is to be displayed. In the case of an integer tag, this means you tell Crimson how many decimal places are to be used, and what units, if any, are to be appended to the value. When you place a tag on a display page, Crimson knows how to format it without you having to do anything further. Similarly, if you decide to change the formatting, and perhaps switch from one set of units to another, you can do this in one place, without having to edit each display page in turn.
- Tags are the key to implementing slave protocols. Crimson treats these protocols as mechanisms for exposing data items within the terminal. This allows the same data to be accessed via multiple ports, so that, for example, a machine setting could be changed by both a local SCADA package, and a similar package working over Ethernet from a remote site. Without tags, there would be nothing to expose, and this mechanism could not be implemented!
- Tags are used within Crimson to implement many advanced features. If you want to use functionality such as alarms, triggers, data logging or the web server, you will have to use tags, period. The formatting data from the tag definition is typically required by all these features, so tags are mandatory for their operation.

In other words, tags will automate many tasks during programming, saving you time. Even if you decide not to use tags, many of the subsequent chapters of this manual refer to concepts discussed in this chapter. You should thus read it thoroughly before proceeding.

CREATING TAGS

To create a tag, either click on one of the buttons displayed when the Tags icon is selected in the left-hand pane of the Data Tags window, or use the new tag buttons on the toolbar. Either way, a new tag will be added to the tag list. To edit the tag's name, select the tag in the left-hand pane, and type in the new name.

Tag names must conform to the following rules…

- Tag names may not contain spaces or punctuation.
- Tag names must start with a letter of the alphabet or an underscore.
- Subsequent characters must be digits, letters or underscores.
- Names must not exceed 24 characters in length.

EDITING TAGS

When a tag is selected in the left-hand pane, the right-hand pane will change to display a number of tabs, each of which shows certain properties of the tag. Depending on the family and type of the tag, different tabs may be present, and each tab may contain different fields.

No matter what kind of tag is selected, the first tab in the right-hand pane is always the Data tab. This tab contains fields which indicate what data the tag is to represent, and how that data is to be stored—and perhaps transferred to or from a remote device. The exact contents of the tab will vary according to the family and type of the tag in question.

The second tab is always the Format tab. This indicates how the data in the tag is to be formatted when shown on the operator panel's display, or when presented to a user via any other mechanism, such as a web page. The Format tab will take the same form for all tags of the same type, such that all integer tags, for example, share the same set of properties.

The balance of the tabs define alarms and triggers for the tag. These are not included for string tags or for arrays. Alarms are used to detect a condition that needs to be brought to the operator's attention, or simply to log the fact that the condition has occurred. Triggers operate in a similar way, but instead of recording the condition, they are used to execute an action.

EDITING PROPERTIES

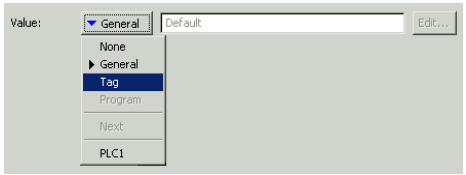
Most properties are edited in ways that are self-evident to anyone who has used a Windows operating system. For example, you may be required to enter a numeric value, or to select an item from a drop-down list. Certain types of property, though, provide more complex editing options, and these are described below.

EXPRESSION PROPERTIES

Expression properties are capable of being set to…

- A constant value.
- The contents of a data tag.
- The contents of a register in a remote communications device.
- A combination of such items linked together using various math operators.

In its default state, the arrowed button immediately after the label of the property shows that the field is in General mode, and the edit box to the right of the button shows a grayed-out string which indicates the default behavior of the property.



If you are familiar with Crimson's expression syntax—a complete description of which can be found in the Writing Expressions section—you can edit the property by typing an expression directly into the edit box. Most users, though, will choose to press the arrowed button and select from the menu of options that is presented...

- Selecting *Tag* will display a dialog box containing a list of data tags. You can select the tag that you want to be used to control this property. In some cases, you will also be given the chance to create a new tag and define its basic

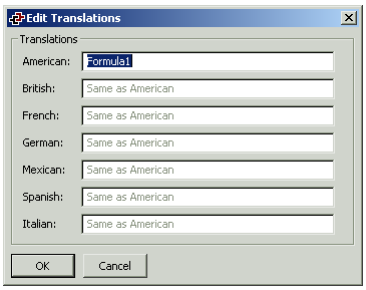
properties. This is not available when editing properties that belong directly to other data tags, as it is otherwise too easy to forget which tag you're editing!

- Selecting a *device name* will display a dialog box allowing you to choose a register within that remote communications device. The various communications devices are listed at the end of the menu in the order in which they were created.
- Selecting *Next* will set the property to be equal to the register which follows the last selected register within the last selected device. For example, if you have used the *device name* option to set a previous property to **N7:10** of PLC1, selecting *Next* will set the current property to **N7:11** of the same device.

TRANSLATABLE STRINGS

Crimson databases are designed to support multi-lingual operation, whereby any string which will be presented to the user of the operator panel is capable of being displayed in one of many different languages. To allow you to define these translations, properties which contain such strings have a button labeled Translate to their right-hand side.

To enter the translations, click the button and the following dialog box will appear…



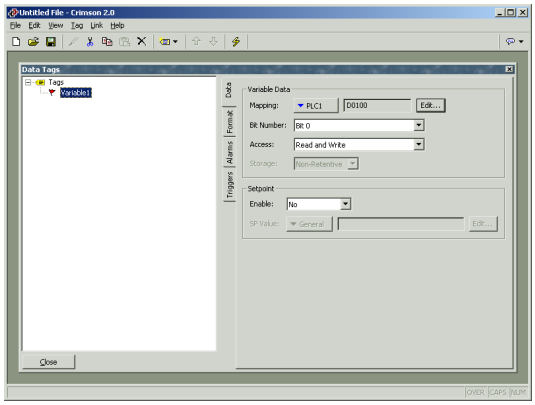
If you do not enter text for a particular language, and that language is subsequently selected by the operator, Crimson will use the American version by default. For information on how to configure a key or a menu to select a different language, refer to the User Interface section.

EDITING FLAG TAGS

You will recall that flag tags represent a true or false value. The following sections describe the various tabs that are displayed on the right-hand side of the Data Tags window when editing one of the various kinds of flag tags.

THE DATA TAB (VARIABLES)

The Data tab of a flag variable contains the following properties...



- The *Mapping* property is used to specify if the variable is to be mapped to a register in a remote device, or if it exists only within the terminal. If you press the arrow button and select a device name from the resulting menu, you will be presented with a dialog box which will allow a PLC register to be selected.

- The *Bit Number* property is used when a flag variable is mapped to a PLC register which contains more than a single bit of information. The property is then used to indicate which bit within the register is to be accessed by the tag.

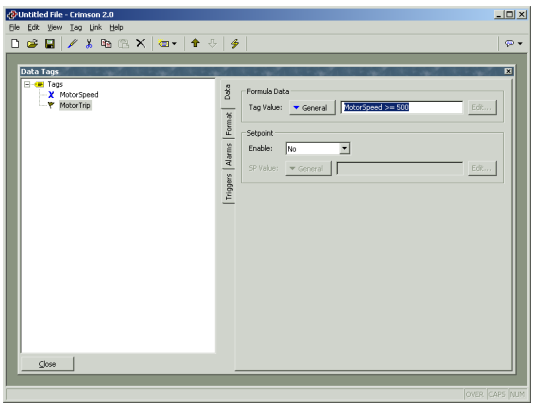
- The *Access* property is used to specify what sort of data transfers should be performed for a mapped variable. You may indicate that data is to be both read and written, or just read or written as appropriate. Write-only tags can be used to avoid unnecessary read operations on data which can only be changed by the terminal. They will typically be set to retentive as their value cannot be obtained from the PLC, and must therefore be stored by the terminal.

- The *Storage* property is used to indicate whether Crimson should allocate FLASH memory within the panel in order to retain the value of the tag during power-down. Mapped tags which are not write-only cannot be set to retentive, as their values will in any case be read from the PLC, and it does not therefore make sense to waste local storage to retain data which will be overwritten.

- The *Setpoint* properties are used to indicate whether a setpoint will be specified for this tag, and what that setpoint will be. Setpoints are used by certain alarm modes, and allow the actual state of a tag to be compared to its intended state. For example, a tag which represents the state of an input from a speed switch for a motor might have the motor's control output specified as a setpoint. This allows an alarm to be programmed to activate if the motor fails to start.

THE DATA TAB (FORMULAE)

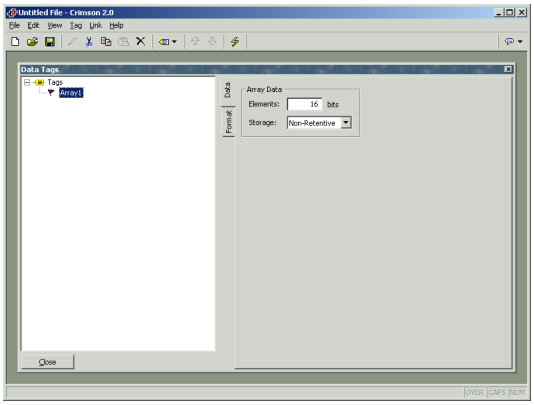
The Data tab of a flag formula contains the following properties...



- The *Tag Value* property is used to specify the value that is to be represented by this tag. It is typically set to a logical combination of other tags or PLC registers, or to a comparison between numeric values. In the example shown above, the tag is configured to be true when a motor speed exceeds a certain value.
- The *Setpoint* properties are as described for flag variables.

THE DATA TAB (ARRAYS)

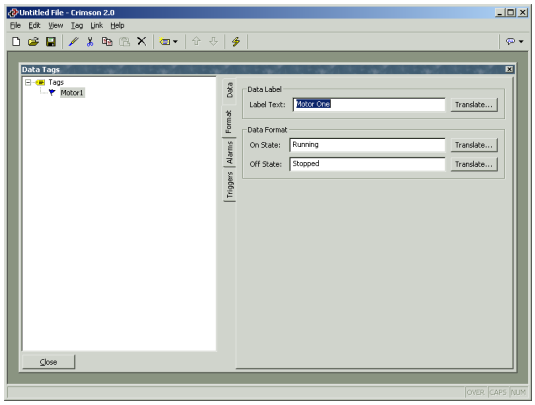
The Data tab of a flag array contains the following properties...



- The *Elements* property is used to indicate how many data items the array should hold. Array elements are referred to using square brackets, such that **`Array[0]`** is the first element, and **`Array[n-1]`** is the last element, where **`n`** is equal to the value entered for this property.
- The *Storage* property is as described for flag variables.

THE FORMAT TAB

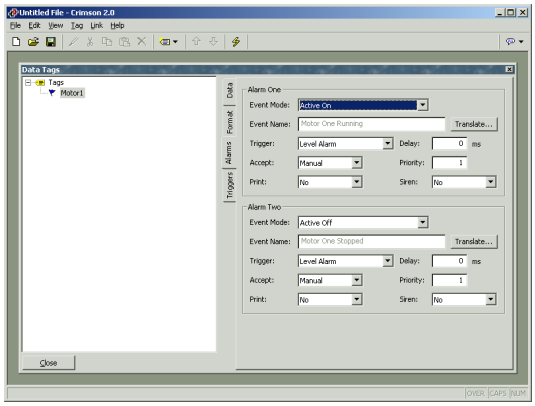
The Format tab of a flag tag contains the following properties...



- The *Label Text* property is used to specify the label that can be shown next to this tag when including the tag on a display page. The label differs from the tag name, in that the former can be translated for international applications, while the latter remains unchanged and is never shown to the user of the panel.
- The *On State* and *Off State* properties are used to specify the text to be displayed when the tag contains a non-zero and zero value, respectively. When you enter the text for the on state, Crimson will attempt to generate corresponding text for the off state by referring both to previously-created flag tags, and to its internal list of common antonymic pairs.

THE ALARMS TAB

The Alarms tab of a flag variable or formula contains the following properties...



- The *Event Mode* property is used to indicate the logic that will be used to decide whether the alarm should activate. The tables below list the available modes.

MODE	ALARM WILL ACTIVATE WHEN...
Active On	The tag is true.
Active Off	The tag is false.

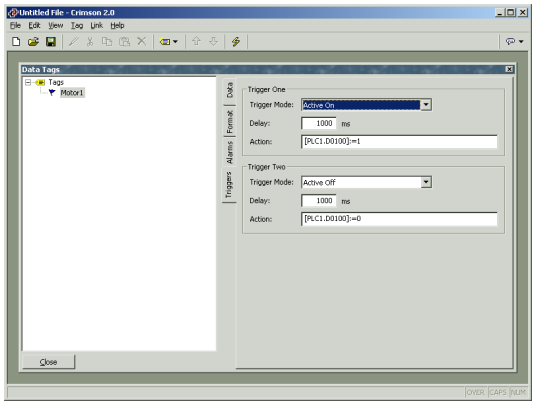
The following modes are only available when a setpoint is defined…

MODE	ALARM WILL ACTIVATE WHEN…
Not Equal to SP	The tag does not equal its setpoint.
Off When SP On	The tag does not respond to an ON setpoint.
On When SP Off	The tag does not respond to an OFF setpoint.
Equal to SP	The tag equals its setpoint.

- The *Event Name* property is used to define the name that will be displayed in the alarm viewer or in the event log as appropriate. Crimson will suggest a default name based upon the tag's label, and the event mode that has been selected.
- The *Trigger* property is used to indicate whether the alarm should be edge or level triggered. In the former case, the alarm will trigger when the condition specified by the event mode first becomes true. In the latter case, the alarm will continue in the active state while the condition persists. This property can also be used to indicate that this alarm should be used as an event only. In this case, the alarm will be edge triggered, but will not result in an alarm condition. Rather, an event will be logged to the G3's internal memory.
- The *Delay* property is used to indicate how long the alarm condition must exist before the alarm will become active. In the case of an edge triggered alarm or event, this property also specifies the amount of time for which the alarm condition must no longer exist before subsequent reactivations will result in a further alarm being signaled. As an example, if an alarm is set to activate when a speed switch indicates that a motor is not running even when the motor has been requested to start, this property can be used to provide the motor with time to run-up before the alarm is activated.
- The *Accept* property is used to indicate whether the user will be required to explicitly accept an alarm before it will no longer be displayed. Edge triggered alarms must always be manually accepted.
- The *Priority* property is used to control the order in which alarms are displayed by Crimson's alarm viewer. The lower the numerical value of the priority field, the nearer to the top the alarm will be displayed.
- The *Print* property is for future expansion.
- The *Siren* property is used to indicate whether or not the activation of this alarm should also activate the G3 panel's internal sounder. While the sounder is active, the panel's display will also flash to better draw attention to the alarm condition.

THE TRIGGERS TAB

The Triggers tab of a flag variable or formula contains the following properties...



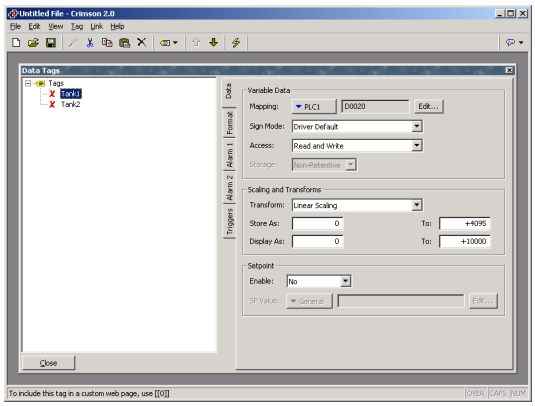
- The *Trigger Mode* property is as described for the Alarms tab.
- The *Delay* property is as described for the Alarms tab.
- The *Action* property is used to indicate what action should be performed when the trigger is activated. Refer to the Writing Actions section for a description of the syntax used to define the various actions that Crimson supports.

EDITING INTEGER TAGS

You will recall that integer tags represent a 32-bit signed value. The following sections describe the various tabs that are displayed on the right-hand side of the Data Tags window when editing one of the various kinds of integer tags.

THE DATA TAB (VARIABLES)

The Data tab of an integer variable contains the following properties...



- The *Mapping* property is used to specify if the variable is to be mapped to a register in a remote device, or if it exists only within the terminal. If you press the arrow button and select a device name from the resulting menu, you will be presented with a dialog box which will allow a PLC register to be selected.

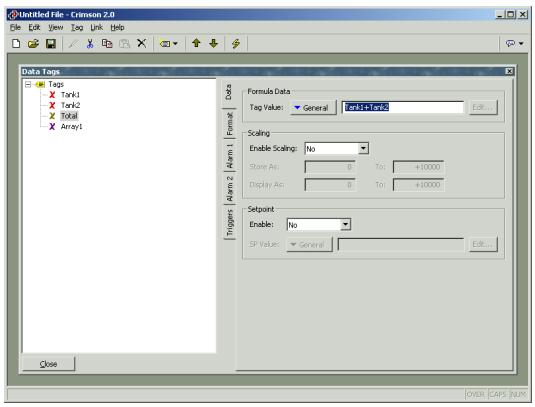
- The *Sign Mode* property is used to override the default behavior of the comms driver when reading 16-bit values from a remote device. The driver will normally make a decision about whether to treat these values as signed or unsigned, based upon how the data is normally used within the device. If you want to override this decision, set this property as required.
- The *Access* property is as described for flag variables.
- The *Storage* property is as described for flag variables.
- The *Scaling and Transforms* properties are used to modify the data value as it is read and written from the remote device. When the linear scaling mode is selected, the *Store As* range indicates the upper and lower bounds of the variable within the PLC, while the *Display As* range indicates the corresponding values as they will be presented to the operator. The other modes are as follows…

MODE	DESCRIPTION
BCD to Binary	The BCD value is converted to binary.
Binary to BCD	The binary value is converted to BCD.
Swap Bytes in Word	The lower two bytes of the value are swapped.
Swap Bytes in Long	All four bytes of the value are swapped.
Swap Words	The upper and lower words of the value are swapped.
Reverse Bits in Byte	Bits 0 through 7 of the value are reversed.
Reverse Bits in Word	Bits 0 through 15 of the value are reversed.
Reverse Bits in Long	Bits 0 through 31 of the value are reversed.
Invert Bits in Byte	Bits 0 through 7 of the value are inverted.
Invert Bits in Word	Bits 0 through 15 of the value are inverted.
Invert Bits in Long	Bits 0 through 31 of the value are inverted.

- The *Setpoint* properties are used to indicate whether a setpoint will be specified for this tag, and what that setpoint will be. Setpoints are used by certain alarm modes, and allow the state of a tag to be compared to its intended state. For example, a tag which represents the temperature of a vessel might have a setpoint that indicates the required temperature. This will allow an alarm to activate if the vessel strays beyond a certain distance from its target.

THE DATA TAB (FORMULAE)

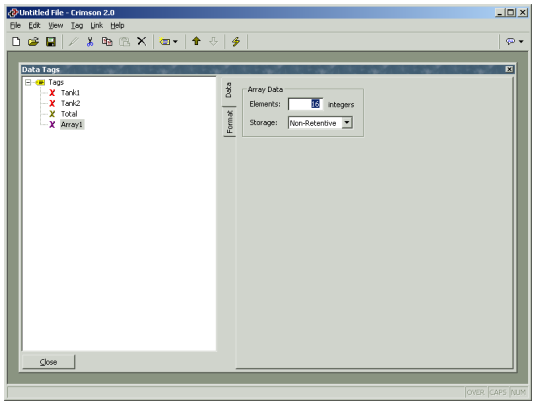
The Data tab of an integer formula contains the following properties...



- The *Tag Value* property is used to specify the value represented by this tag. It is typically set to a combination of other tags, linked together using math operators. In the example above, the tag is set to be equal to the sum of two tank levels, therefore indicating the total amount of feedstock available.
- The *Scaling and Transforms* properties are as described for integer variables.
- The *Setpoint* properties are as described for integer variables.

THE DATA TAB (ARRAYS)

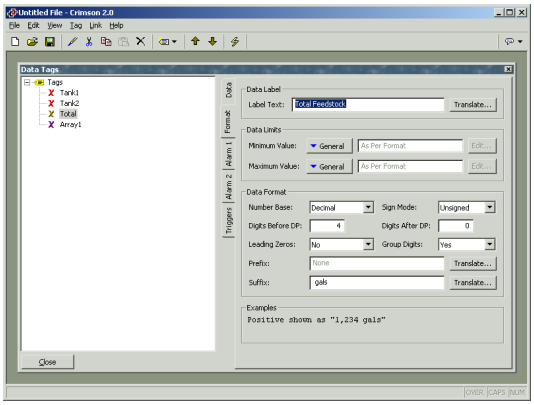
The Data tab of an integer array contains the following properties...



All of these properties are as described for flag arrays.

THE FORMAT TAB

The Format tab of an integer tag contains the following properties...



- The *Label Text* property is used to specify the label that can be shown next to this tag when including the tag on a display page. The label differs from the tag name, in that the former can be translated for international applications, while the latter remains unchanged and is never shown to the user of the panel.

- The *Minimum Value* and *Maximum Value* properties are used to define the limits used for data entry, and to provide similar limits for the various graphical primitives which need to know the bounds within which the tag may vary, such as when scaling a tag's value for display as a bar-graph.

- The *Number Base* property is used to indicate whether the tag should be displayed in decimal, hexadecimal, binary or octal. Decimal values may be signed or unsigned, while all other number bases imply unsigned operation.

- The *Sign Mode* property is used to indicate whether or not a sign should be prefixed to the tag's value. If a hard sign is selected, either a positive or a negative sign will be prefixed as appropriate. If a soft sign is selected, a positive sign will not be shown, but a space will be prefixed instead.

- The *Digits Before DP* property is used to indicate how many digits should be shown before the decimal place, or, if no digits are to be shown after the decimal place, to indicate how many digits should be shown in total.

- The *Digits After DP* property is used to indicate how many digits should be shown after the decimal place. Somewhat obviously, decimal places are not supported if a number base other than decimal has been selected!

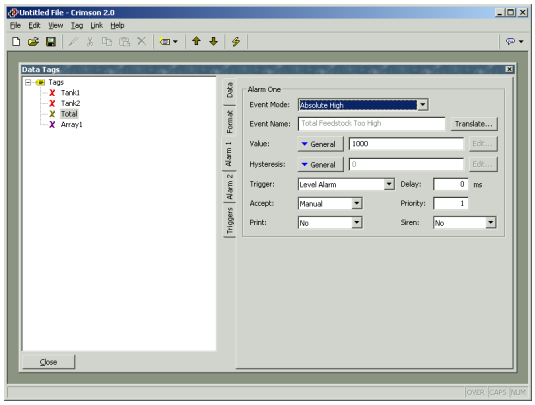
- The *Leading Zeroes* property is used to indicate whether zeros at the beginning of a number should be shown, or replaced with spaces.

- The *Group Digits* property is used to indicate whether decimal values should have the digits before the decimal place grouped in threes, and separated with commas. Similar separation is performed on other number bases, using groupings and separators appropriate to the selected radix.

- The *Prefix* property is used to specify a translatable string that will be displayed in front of the numeric value. This is typically used to indicate units of measure.
- The *Suffix* property is used to specify a translatable string that will be displayed after the numeric value. This is also typically used to indicate units of measure.

THE ALARM TABS

Each Alarm tab of an integer variable or formula contains the following properties...



- The *Event Mode* property is used to indicate the logic that will be used to decide whether the alarm should activate. The tables below list the available modes.

MODE	ALARM WILL ACTIVATE WHEN...
Data Match	The value of the tag is equal to the alarm's *Value*.
Data Mismatch	The value of tag is not equal to the alarm's *Value*.
Absolute High	The value of the tag exceeds the alarm's *Value*.
Absolute Low	The value of the tag falls below the alarm's *Value*.

The following modes are only available when a setpoint is defined…

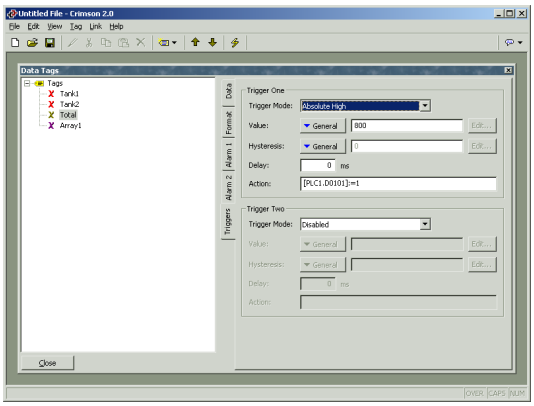
MODE	ALARM WILL ACTIVATE WHEN...
Deviation High	The value of the tag exceeds the tag's *Setpoint* by an amount equal to or greater than the alarm's *Value*.
Deviation Low	The value of the tag falls below the tag's *Setpoint* by an amount equal to or greater than the alarm's *Value*.
Out of Band	The tag moves outside a band equal in width to twice the alarm's *Value* and centered on the tag's *Setpoint*.
In Band	The tag moves inside a band equal in width to twice the alarm's *Value* and centered on the tag's *Setpoint*.

- The *Value* property is used to define either the absolute value at which the alarm will be activated, or the deviation from the setpoint value. The exact interpretation depends on the event mode as described above.

- The *Hysteresis* property is used to prevent an alarm from oscillating between the on and off states when the process is near the alarm condition. For example, for an absolute high alarm, the alarm will become active when the tag exceeds the alarm's value, but will only deactivate when the tag falls below the value by an amount greater than or equal to the alarm's hysteresis. Remember that the property always acts to maintain an alarm once the alarm is activated, and not to modify the point at which the activation occurs.
- The remainder of the properties are as described for the Alarms tab of flag tags.

THE TRIGGERS TAB

The Triggers tab of an integer variable or formula contains the following properties...



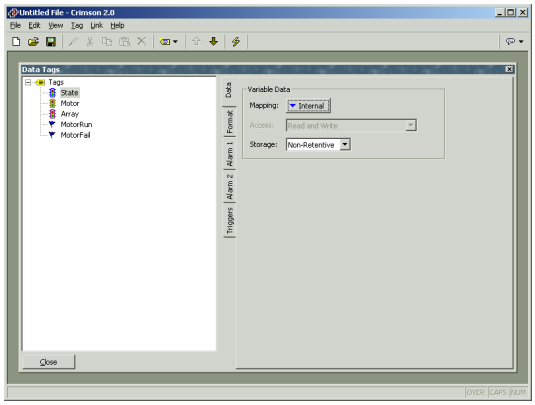
- The *Trigger Mode* property is as described for the Alarm tabs.
- The *Delay* property is as described for a flag tag's Alarms tab.
- The *Action* property is used to indicate what action should be performed when the trigger is activated. Refer to the Writing Actions section for a description of the syntax used to define the various actions that Crimson supports.

EDITING MULTI TAGS

You will recall that multi tags represent a 32-bit signed value, but are used to select between one of a number of text strings. The following sections describe the various tabs that are displayed on the right-hand side of the Data Tags window when editing a multi tag.

THE DATA TAB (VARIABLES)

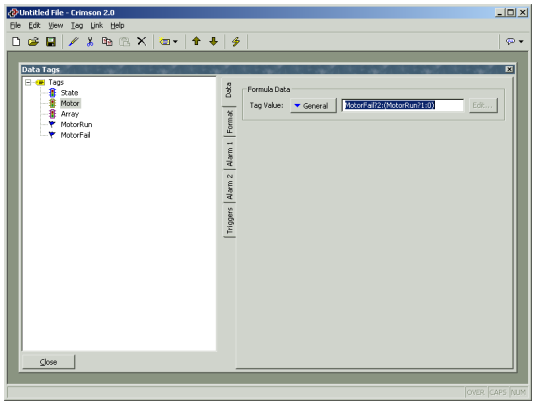
The Data tab of a multi variable contains the following properties...



- The *Mapping* property is used to specify if the variable is to be mapped to a register in a remote device, or if it exists only within the terminal. If you press the arrow button and select a device name from the resulting menu, you will be presented with a dialog box which will allow a PLC register to be selected.
- The *Access* property is as described for flag variables.
- The *Storage* property is as described for flag variables.

THE DATA TAB (FORMULAE)

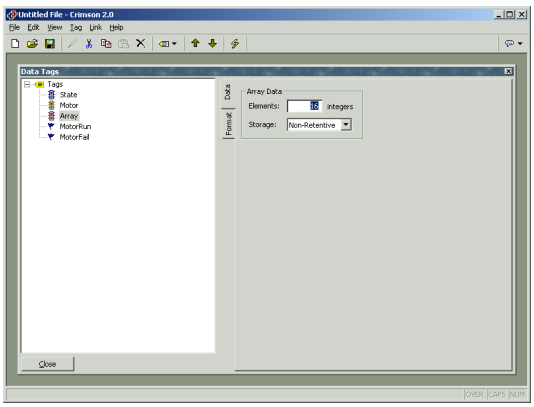
The Data tab of a multi formula contains the following properties...



- The *Tag Value* property is used to specify the value represented by this tag. It is typically set to a combination of other tags, linked together using math operators. In the example above, the tag is set equal to a value of one, two or three, depending on the state of three different flags. For more information on the `?:` operator used in this example, refer to the Writing Expressions section.

THE DATA TAB (ARRAYS)

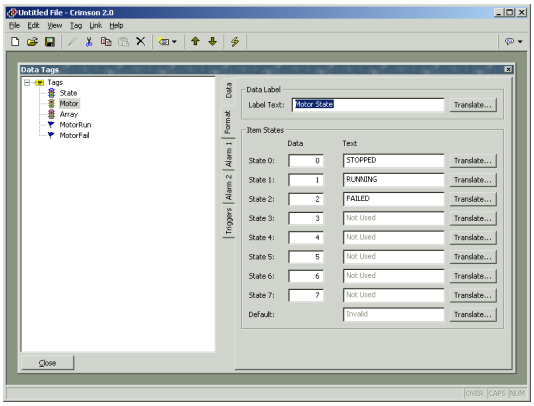
The Data tab of a multi array contains the following properties...



All of these properties are as described for flag arrays.

THE FORMAT TAB

The Format tab of a multi tag contains the following properties...



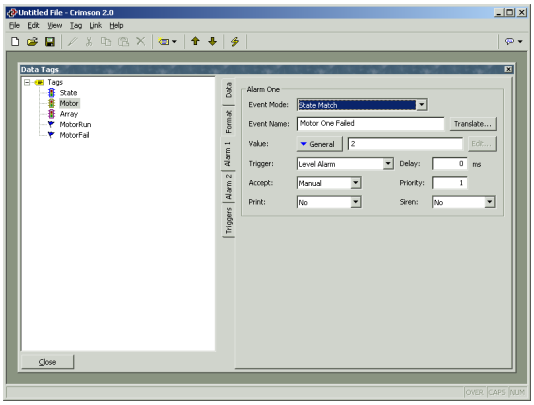
- The *Label Text* property is used to specify the label that can be shown next to this tag when including the tag on a display page. The label differs from the tag name, in that the former can be translated for international applications, while the latter remains unchanged and is never shown to the user of the panel.

- The *Item States* properties are used to define up to eight values which represent different states of the tag. Each state has an integer value associated with it, and a text string to indicate what should be displayed when the tag holds that value. At least two states must be defined, but the balance may be left in their default condition if they are not needed.

- The *Default* property is used to define the text to be displayed if the tag holds a value other than one of those listed in the item states.

THE ALARM TABS

Each Alarm tab of a multi variable or formula contains the following properties...



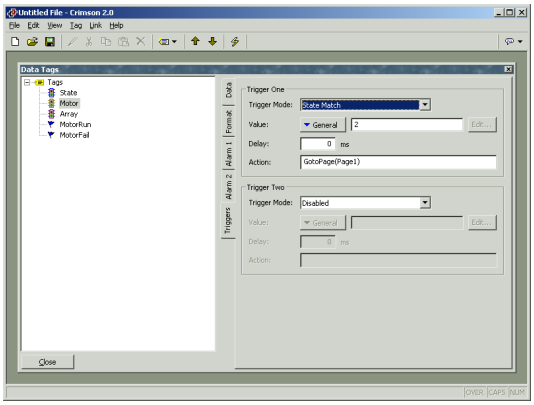
- The *Event Mode* property is used to indicate the logic that will be used to decide whether the alarm should activate. The table below lists the available modes.

MODE	ALARM WILL ACTIVATE WHEN...
State Match	The value of the tag is equal to the alarm's *Value*.
State Mismatch	The value of tag is not equal to the alarm's *Value*.

- The *Value* property is used to define the comparison data for the alarm.
- The remainder of the properties are as described for the Alarms tab of flag tags.

THE TRIGGERS TAB

The Triggers tab of a multi variable or formula contains the following properties...



- The *Trigger Mode* property is as described for the Alarm tabs.
- The *Delay* property is as described for a flag tag's Alarms tab.
- The *Action* property is used to indicate what action should be performed when the trigger is activated. Refer to the Writing Actions section for a description of the syntax used to define the various actions that Crimson supports.

EDITING REAL TAGS

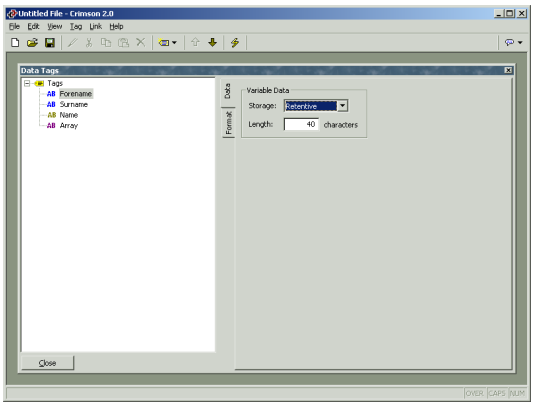
You will recall that real tags represent a single-precision floating-point value. All the tabs displayed for real tags are exactly analogous to those displayed for integer tags, with the exception that data entered for items such as the value and hysteresis properties of alarms and triggers may contain decimals. You are thus referred to the sections on integer tags.

EDITING STRING TAGS

You will recall that string tags represent an item of text, this being made up of a number of individual characters. The following sections describe the various tabs that are displayed on the right-hand side of the Data Tags window when editing one of the various string tags.

THE DATA TAB (VARIABLES)

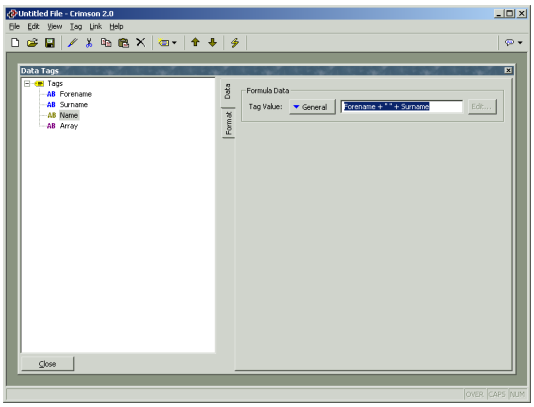
The Data tab of a string variable contains the following properties...



- The *Storage* property is as described for flag variables.
- The *Length* property is used to indicate how many characters of storage should be allocated for this string. A value need only be entered if you have configured the variable for retentive storage. Strings that are kept in the terminal's RAM and not committed to FLASH have no practical limit on their length.

THE DATA TAB (FORMULAE)

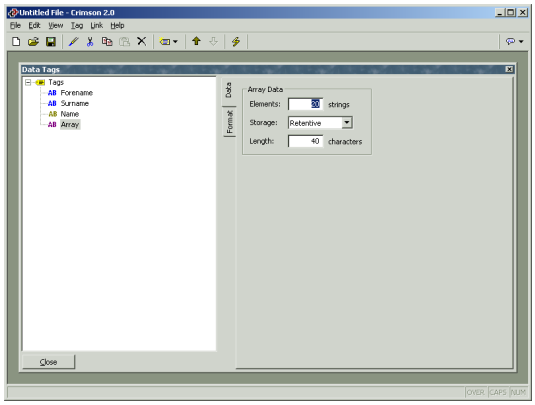
The Data tab of a string formula contains the following properties...



- The *Tag Value* property is used to specify the value represented by this tag. It is typically set to a combination of other tags, linked together using math operators or functions. In the example above, the tag is set equal to the combination of two strings variables, separated by a space. For more information on the operators and functions that can be used with strings, refer to the Writing Expressions section and the Function Reference at the end of this document.

THE DATA TAB (ARRAYS)

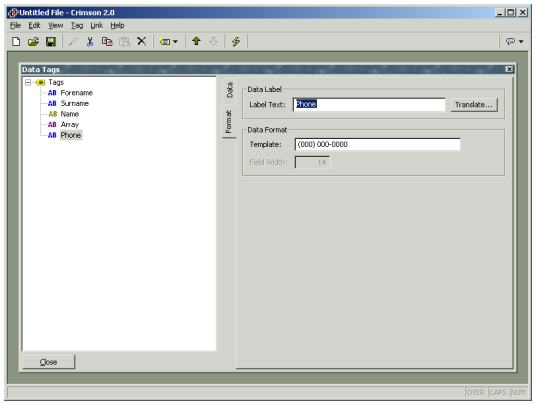
The Data tab of a string array contains the following properties...



- The *Elements* and *Storage* properties are as described for flag arrays.
- The *Length* property is as described for string variables.

THE FORMAT TAB

The Format tab of a string tag contains the following properties...



- The *Label Text* property is used to specify the label that can be shown next to this tag when including the tag on a display page. The label differs from the tag name, in that the former can be translated for international applications, while the latter remains unchanged and is never shown to the user of the panel.
- The *Template* property is used to provide a “picture” of the string, thereby indicating what kind of characters can occur in each position. If a template is

specified, data entry will be limited such that only the correct kind of character can be selected for each character in the string. The table below shows the meaning of the various special characters that can be included in a template…

Character In Template	Permitted Characters A-Z	a-z	0-9	Space	Misc
A	Yes	-	-	-	-
a	Yes	Yes	-	-	-
S	Yes	-	-	Yes	-
s	Yes	Yes	-	Yes	-
N	Yes	-	Yes	-	-
n	Yes	Yes	Yes	-	-
M	Yes	-	Yes	Yes	-
m	Yes	Yes	Yes	Yes	-
0	-	-	Yes	-	-
X	Yes	Yes	Yes	Yes	Yes

The additional characters referred to by the "Misc" column are…

```
,.:;+-=!?%/$
```

Characters not included in the table are copied verbatim to the display.

For example, to allow entry of a US telephone number, use a template of…

```
(000) 000-0000
```

The parentheses, the space and the dash will all be included when the field is displayed, but only the 10 digits indicated by the '0' characters will be stored in the string. Similarly, if data entry is enabled for a field using this template, the cursor will skip the various non-numeric positions when moving left or right, and will only allow numeric characters to be entered for those positions which can be selected.

- The *Length* property is used in lieu of the template to indicate how many characters should be reserved on a page when displaying this string. If a string variable is marked as retentive, it makes sense for this property to be equal to the length entered on the *Data* tab, but this is not obligatory, as you may want to allocate more or less space on the display for layout purposes.

MORE THAN TWO ALARMS

If your application requires more than two alarms (or indeed triggers) for a tag, define a formula to be equal in value to the primary tag, and set the extra alarms on the alias. For example, if you have a variable called **`Level`** which is mapped to **`N7:100`** in a PLC, and you need to create a third alarm for that tag, create a variable called, say, **`LevelAlias`** and set its value property to **`Level`**. You can then set additional alarms on this alias tag.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

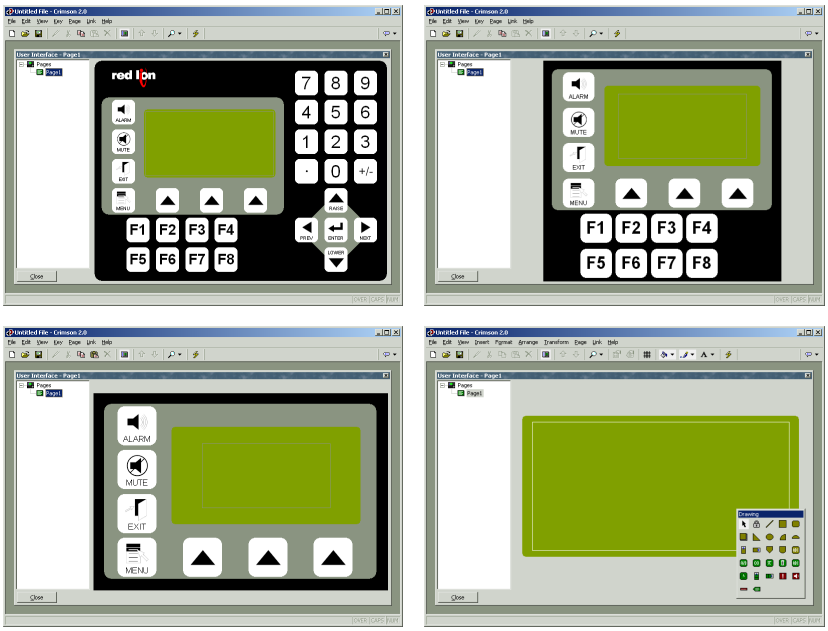
- Crimson's Data Tags window is used to perform all of the various functions that were previously implemented using the Named Data window, the Alarm Scanner, and the Trigger Table.
- Crimson does not have the concept of constants as a separate tag family. Edict used constants to help it perform certain optimizations that Crimson is now able to perform automatically. Constants can thus be implemented using formulae.
- Crimson associates alarms and triggers with tags, rather than allowing them to be defined on the basis of arbitrary expressions. If you need to have an alarm or trigger monitor a general expression, define a formula to be equal to that expression, and set the alarm and/or trigger on that tag instead.

CONFIGURING THE USER INTERFACE

Now that you have configured your communications, and created data tags for the items that you wish to display, you can create display pages to allow the user to view or edit these data items. These pages are manipulated by selecting the User Interface icon from the main screen.

CONTROLLING THE VIEW

By default, the User Interface window shows the entire front panel of the G303, including the display and all the available keys. If you want to allocate more of your PC's screen to show the G303's display, you can use the four different zoom levels as shown below…



As you can see, at each level, fewer keys are shown, and more of the window is allocated to the display itself. The zoom level can be controlled from the View menu, by using the magnifying glass icon, or by pressing the **Alt** key together with the digits **1** through **4**.

OTHER VIEW OPTIONS

As well as controlling the zoom, the View menu contains the following options…

- The *Page List* command can be used to show or hide the left-hand pane of the User Interface window. If the page list is disabled, even more space is made available for editing the display. The **F4** key toggles the page list on and off.
- The *Hold Aspect* command can be used to control whether or not Crimson attempts to maintain the aspect ratio of the display. If aspect holding is enabled, a figure that would appear as, say, a circle on the G303 will appear as a perfect circle on your PC. If this mode is not selected, Crimson can expand the display page to use more of the PC's screen, but at the expense of some distortion.

Other options are available during page editing, and are described below.

USING THE PAGE LIST

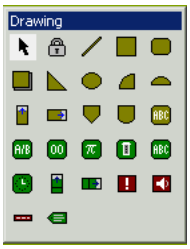
To create, rename or delete display pages, click on the left-hand pane of the User Interface window. The various commands on the Page menu can then be used to make the desired changes. Alternatively, right-click on the required display page, and select from the menu.

To select a page, either click on the page in the page list, or use the up and down arrows in the toolbar. Alternatively, you can use the **`Alt+Left`** and **`Alt+Right`** key combinations to move up and down the list as required. These keys will work no matter which pane is selected.

DISPLAY EDITOR TOOLBOXES

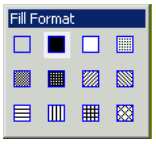
To edit the contents of a display page, first select the page as described above. Then, click on the green rectangle that represents the G3's display. A white rectangle will appear around the display to indicate that it has been selected, and a number of toolboxes will appear.

THE DRAWING TOOLBOX



The drawing toolbox is used to add various elements, known as primitives, to the display page. The first two icons control the insertion mode, while the balance of the icons represent individual primitives. The primitives shown in yellow are basic geometric and animation items, while the ones shown in green are rich primitives that use formatting and other information from a data tag to control their operation. The primitives shown in red are system items, such as the active alarm viewer. All of the commands contained in the toolbox can also be accessed via the Insert menu.

THE FILL FORMAT TOOLBOX

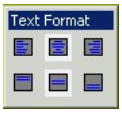


The fill format toolbox is used to control the pattern that will be used to fill a display primitive. If one or more primitives is selected, clicking on a fill pattern will change all the selected items to use that pattern. If nothing is currently selected, clicking on a pattern will set the default pattern for newly-created primitives. The various options can also be accessed via the Format menu, or via the paint-can icon on the toolbar.

THE LINE FORMAT TOOLBOX

The line format toolbox is used to control the color that will be used to draw an outline around a display primitive. If one or more primitives is selected, clicking on a line color will change all the selected items to use that color. If nothing is currently selected, clicking on a color will set the default outline color for newly-created primitives. The various options can also be accessed via the Format menu, or via the paintbrush icon on the toolbar.

THE TEXT FORMAT TOOLBOX



The text format toolbox is used to control the horizontal and vertical alignment of primitives which contain text elements. If one or more such primitives is selected, clicking on an icon will change all the selected items to use the selected

justification. If nothing is currently selected, clicking on an icon will set the default format for newly-created primitives. The various options can also be accessed via the Format menu.

THE FOREGROUND TOOLBOX

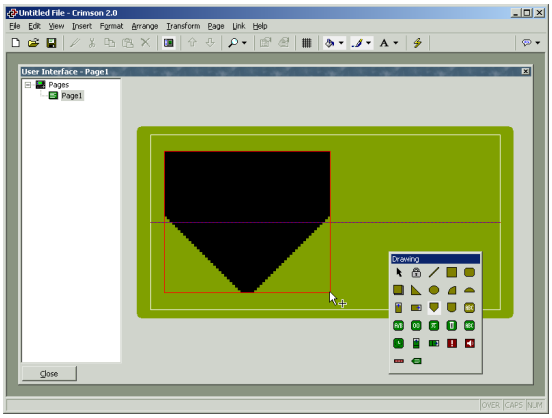
The foreground toolbox is used to control the foreground color for primitives which contain text elements. If one or more such primitives is selected, clicking on an icon will change all the selected items to use the selected color. If nothing is currently selected, clicking on an icon will set the default color for all newly-created primitives. The various options can also be accessed via the Format menu.

THE BACKGROUND TOOLBOX

The background toolbox is used to control the foreground color for primitives which contain text elements. If one or more such primitives is selected, clicking on an icon will change all the selected items to use the selected color. If nothing is currently selected, clicking on an icon will set the default color for all newly-created primitives. The various options can also be accessed via the Format menu.

ADDING DISPLAY PRIMITIVES

To add a display primitive to a page, click on the required icon in the drawing toolbox, or select the required option from the Insert menu. The mouse cursor will change to an arrow with a crosshair at its base, and you will then be able to drag-out the required position of the primitive within the display window…

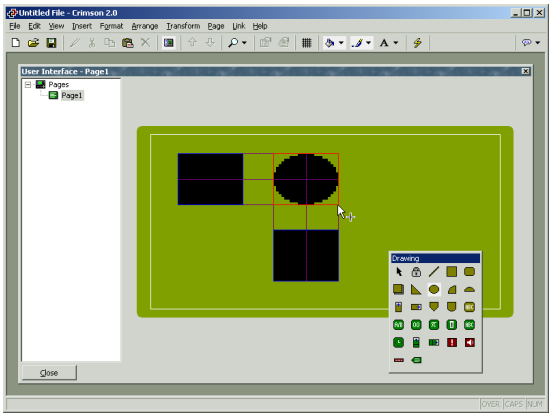


SMART ALIGNMENT

If you have the Smart Align features of the View menu enabled, Crimson will provide you with guidelines to help align a new primitive with existing primitives, or with the center of the display. In the example shown above, the horizontal dotted line indicates that the center of the tank primitive is vertically aligned with the center of the display. With a little practice, this feature can make it very easy to align primitives as they are created, without the need to go back and "tweak" your display pages to get the various figures into alignment.

In the Smart Align example shown below, a newly-created ellipse is being aligned with two existing rectangles. Guidelines are present at both the edges of the figures, and at the center,

showing that both the edges and the centers are aligned. The red rectangle is highlighting the newly-created primitive, while the blue rectangles are highlighting the primitives to which the guidelines have been drawn.



Smart Align is also enabled when primitives are moved or re-sized.

KEYBOARD OPTIONS

While creating a display primitive, the following keyboard options are available…

- Holding down the **`Shift`** key while dragging-out the primitive will cause the primitive to be drawn such that it is centered on the initial mouse position, with one of its corners defined by the current mouse position. (If this doesn't make sense, go ahead and try it—it's a lot easier to see than it is to explain!) This is useful for drawing symmetrical figures centered on an initial point.
- Holding down the **`Ctrl`** key while dragging-out the primitive will keep its horizontal and vertical sizes the same. This is useful when you want to be sure that you draw an exact circle or square using the ellipse or rectangle primitives.

These options are also active when primitives are re-sized.

LOCK INSERT MODE

The padlock icon on the drawing toolbox can be used to add a number of primitives of the same basic type without having to click the toolbox icon for each item in turn. To cancel lock mode, click the padlock icon again, or press the **`Escape`** key. The same operation can be performed by using the Lock Mode command on the Insert menu.

SELECTING PRIMITIVES

To select a display primitive, simply move your mouse pointer over the primitive in question, and perform a left-click. You will notice that while your pointer is hovering over a primitive, a bounding rectangle is drawn in blue to help show what will be selected. When the actual selection is performed, the rectangle will change to red, and handles will appear, so as to allow you to re-size the primitive as required. If you find that the primitive you want to select is hidden below another primitive, press the **`Alt`** key to allow the selection to be made.

To select several primitives, either drag-out a selection rectangle around the primitives you want to select, or select each primitive in turn, holding down the **Shift** key to indicate that you want each primitive to be added to the selection. If multiple primitives are selected, the red rectangle will surround all of the primitives, and the handles can then be used to resize the primitives as a group. The relative size and position of the primitives will be maintained, as long as Crimson can do so without violating minimum size requirements.

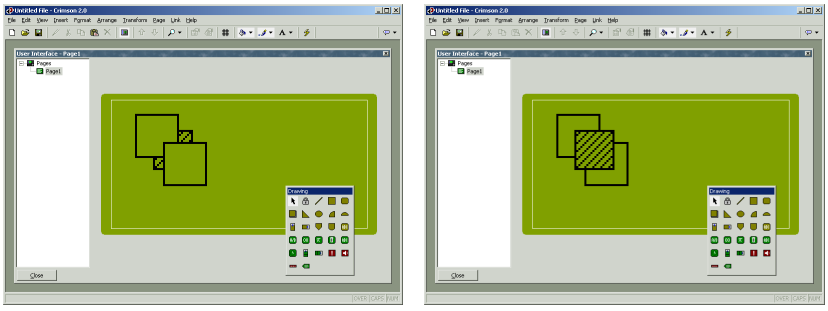
MOVING AND RESIZING

Primitives can be moved by first selecting them, and then by dragging them to the required position on the display page. If Smart Align is turned on, guidelines will appear to help you align the primitives with other items on the page. Holding down **Ctrl** while moving a primitive will leave a copy of the primitive in its original position, thereby allowing duplicates to be created. You can also use the cursor keys to "nudge" the current selection a single pixel in the required direction. Holding down **Ctrl** while nudging will increase the movement of the primitives by a factor of eight.

Primitives can be resized by selecting them, and then by dragging the appropriate handle to the required position. Once again, if Smart Align is turned on, guidelines will appear to help you align the primitives with other items on the page. The **Shift** and **Ctrl** keys can be used to modify the resize behavior as described in the Adding Display Primitives section. Note that Crimson will always constrain resizing operations so as to ensure that primitives stay on the screen, and to make sure that items do not exceed their maximum permitted size, or shrink below the minimum size appropriate to their format.

REORDERING PRIMITIVES

Primitives on a display page are stored in what is known as a z-order. This defines the sequence in which the primitives are drawn, and therefore whether or not a given primitive appears to be in front of or behind another primitive. In the first example below, the hatched square is shown behind the solid squares ie. at the bottom of the z-order. In the second example, it has been moved to the front of the order, and appears in front of the other figures.



To move items in the z-order, select the items, and then use the various commands on the Arrange menu. The Move Forward and Move Backward commands move the selection one step in the indicated direction, while the Move To Front and Move To Back commands move the selection to the indicated end of the z-order. Alternatively, if you have a mouse that is equipped with a wheel, the wheel can be used to move the selection. Scrolling up moves the selection to the back of the z-order; scrolling down moves the selection to the front.

EDITING PRIMITIVES

In addition to the above, primitives can be edited in various ways…

- The various clipboard commands on the Edit menu (eg. Cut, Copy and Paste), or the corresponding toolbar icons, can be used to duplicate items or move them around on a page or between pages. The Duplicate command can be used to perform a Copy operation, immediately followed by a Paste operation. Note that when a Paste is performed, Crimson will offset the newly-pasted item if it will exactly overlay an item of the same type.
- The various formatting properties (eg. fill pattern, outline color, text justification and so on) can be changed by selecting a primitive, and then either clicking the various buttons in the appropriate toolboxes or by using the associated commands on the Format menu. If multiple primitives have been selected, Crimson will apply the changes to all selected primitives.
- The more detailed properties of a primitive can be edited by double-clicking the primitive, or by using the Properties command on the Edit menu. A dialog box will be displayed, allowing all of the primitives to be accessed. The properties associated with each primitive will be described below.

PRIMITIVE DESCRIPTIONS

The sections below describe each primitive found in the drawing toolbox.

THE LINE PRIMITIVE

The *Line* primitive is a line drawn between two points. Its only property is the style of line to be used. In addition to the solid colors shown on the line toolbox, a number of dotted styles can also be accessed via the properties dialog box.

THE SIMPLE GEOMETRIC PRIMITIVES

The *Rectangle* primitive is a rectangle with a defined outline and fill pattern. The fill pattern may be set to No Fill to draw the outline alone, or the outline may be set to None to draw a figure without a border.

The *Round Rectangle* primitive is similar to the rectangle, but has rounded corners. When the primitive is selected, an additional handle appears, allowing the radius of the corners to be edited by dragging the handle from side to side.

The *Shadow* primitive is similar to the rectangle, but with a drop-shadow located to the bottom right of the figure. The primitive is often drawn with no fill pattern, so as to allow it to act as a frame around text primitives.

The *Wedge* primitive is a right-angled triangle located within one quadrant of a bounding rectangle. In addition to the outline and fill properties, the wedge has a property to indicate which quadrant it should occupy.

The *Ellipse* primitive is an ellipse with a defined outline and fill pattern. The fill pattern may be set to No Fill to draw the outline alone, or the outline may be set to None to draw a figure without a border.

The *Ellipse Quadrant* primitive is one quadrant of an ellipse. In addition to the outline and fill properties, the ellipse quadrant has a property to indicate which quadrant it should occupy.

The *Ellipse Half* primitive is one half of an ellipse. In addition to the outline and fill properties, the ellipse half has a property to indicate which of the four possible halves (think about it!) will be drawn.

The properties for these primitives need little further explanation, other than to point out that the quadrant or half rendered by the Wedge, Ellipse Quadrant or Ellipse Half primitives can also be edited via the command found on the Transform menu.

THE TANK PRIMITIVES

The *Conical Tank* primitive is a conical tank with a defined outline and fill pattern. When the primitive is selected, additional handles appear, allowing the exact shape of the tank to be modified by dragging the handles as required.

The *Round Bottomed Tank* primitive is a tank with a defined outline and fill pattern. When the primitive is selected, an additional handle appears, allowing the exact shape of the tank to be modified by dragging the handle as required.

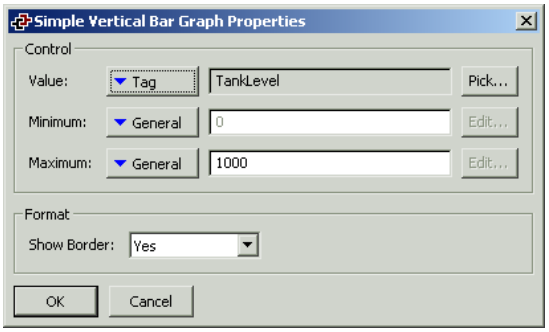
The properties for these primitives need little further explanation.

THE SIMPLE BAR-GRAPH PRIMITIVES

The *Simple Vertical Bar* primitive allows an expression to be drawn as a vertical bar-graph between specified minimum and maximum values. An additional property allows the primitive's border to be displayed or hidden.

The *Simple Horizontal Bar* primitive allows an expression to be drawn as a horizontal bar-graph between specified minimum and maximum values. An additional property allows the primitive's border to be displayed or hidden.

The properties are accessed by double-clicking the primitive…

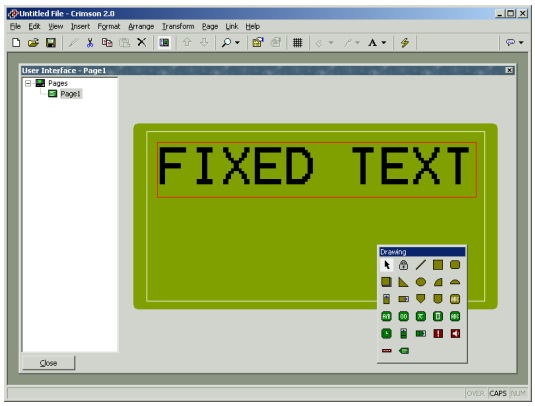


- The *Value* property is used to specify the value to be displayed. In the example given above, the primitive is configured to display the level of a tank.
- The *Minimum* and *Maximum* properties are used to specify the range of values to be shown. In the example above, a range of 0 to 1000 is specified.
- The *Show Border* property is used to display or hide the primitive's border.

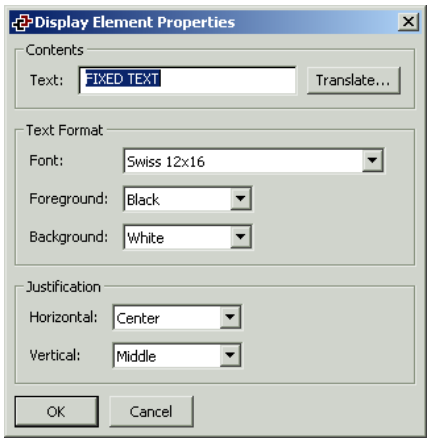
THE FIXED TEXT PRIMITIVE

The *Fixed Text* primitive is used to add unchanging text to a page. The text is displayed in a specified font and color, and with a specified justification. The text can also be translated for international applications.

When the text is created, a cursor will appear, allowing the text to be entered…



Only the US English text can be edited directly. The international versions of the text must be edited via the properties dialog box, which is accessed by selecting the primitive and pressing **`Alt+Enter`**, or by selecting the Properties command from the Edit menu…



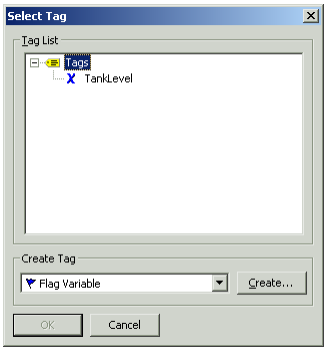
- The *Text* property is used to specify the text to be displayed. As mentioned above, the US English version of the text can also be edited directly on the display page when the primitive is created, or by clicking an existing primitive.

- The *Font* property is used to specify the font to be used. This property can also be edited by using the font button on the toolbar, or by using the Format menu.
- The *Foreground* and *Background* properties are used to specify the colors to be used to draw the text. Obviously, having the same color for both settings will render the text unreadable. Selecting None for the background will create transparent text, allowing underlying primitives to be seen through the letters.
- The *Horizontal* and *Vertical* justification properties are used to indicate where the text should be placed within the bounding rectangle of the primitive. These properties can also be edited via the associated toolbox, or via the Format menu.

THE AUTO TAG PRIMITIVE

The *Auto Tag* primitive allows you to select a tag, and then automatically place the appropriate text primitive on the display. For example, selecting an integer tag will allow insertion of an appropriately-configured integer text primitive.

This is the icon you will use most often for adding tags to a page. It first displays the dialog box shown below to allow tag selection, and then creates one of the five tag text primitives described in the next section. The new primitive will be configured so as to display the tag in question using its label and its formatting properties, as defined when the tag was created.



THE TAG TEXT PRIMITIVES

The tag text primitives are used to display or edit an expression in textual form. Primarily, they are used to display tags, in which case the default format is taken from the Format tab associated with that tag in the Data Tags window. If a non-tag expression is entered—or if you want the formatting to differ from the default values for a tag—the format data can be overridden as required. There is one type of tag text for each tag family…

The *Flag Text* primitive is used to display a true or false condition.

The *Integer Text* primitive is used to display an integer expression.

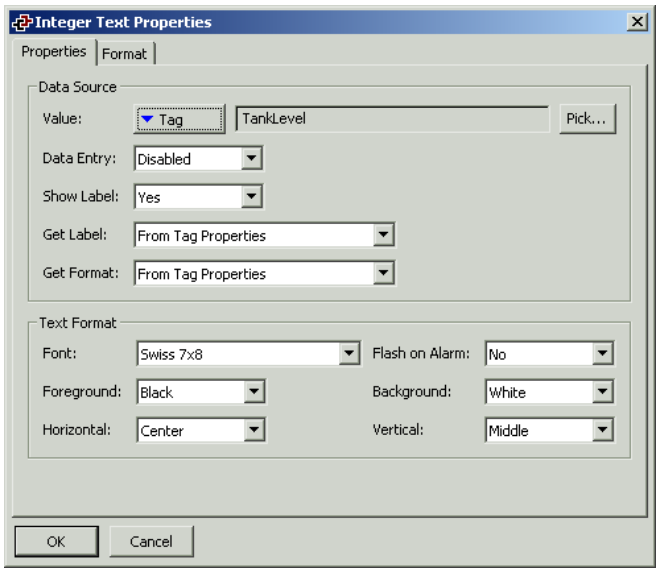
The *Real Text* primitive is used to display a floating-point expression.

The *Multi Text* primitive is used to display a multi-state condition.

The *String Text* primitive is used to display a string expression.

The properties of a tag text primitive are displayed using two tabbed pages.

The first page is more-or-less the same for all five primitive types…



- The *Value* property is used to indicate from where the data for this primitive should be obtained. You may select a tag, a register in a communications device, or an expression which combines a number of such items. The data type of the item must be appropriate to the primitive in question eg. the Value property for an integer text primitive cannot be set equal to a string expression.

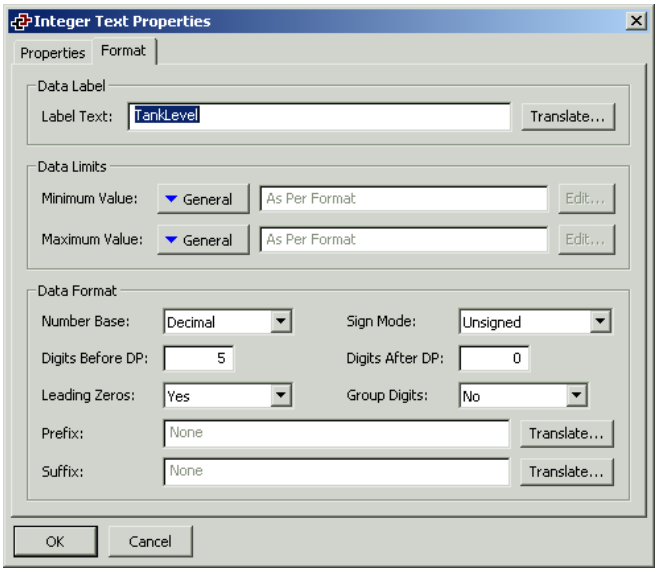
- The *Data Entry* property is used to indicate whether or not you want the user of the operator interface panel to be able to change the underlying value via this primitive. For data entry to be enabled, the expression entered for the value property must be capable of being changed. For example, if a formula is entered, data entry will not be permitted.

- The *Show Label* property is used to indicate whether or not you want the primitive to include a label to identify the data being displayed. If this property is set to yes, the label will be left-justified within the primitive's bounding rectangle, while the data itself will be right-justified. If this property is set to no, the Horizontal Justification property will be used to locate the data within the field. Note that this property can be edited via the Field Label commands on the

Format menu. When no primitive is selected, these commands can also be used to set the default value for newly-created primitives.

- The *Get Label* property is used to indicate from where the label text should be obtained. The options presented depend on what was entered for the value property. If a tag has been selected, you will be given the option of using the tag's default label, or entering a new label on the Format tab of the dialog box. If something else has been selected, you will only have the second option.
- The *Get Format* property is used to indicate from where the formatting information for this primitive should be obtained. The options presented depend on what was entered for the value property. If a tag of the correct data type has been selected, you will be given the option of using the tag's default formatting, or entering modified information on the Format tab of the dialog box. If something else has been selected, you will only have the second option.
- The *Flash on Alarm* property is used to indicate whether or not you want the text on the G3's display to flash if the tag entered in the value property is currently in an alarm state. This property is not available for string text primitives, or for those primitives which have a non-tag value defined for the value property.
- The balance of the properties control the font, colors and justification to be used when drawing the primitive. These properties require no further explanation.

The second page varies according to the primitive in question, and displays the same information as the Format tab of the associated tag type. Different sections of the page will be enabled according to the settings provided for the Get Label and Get Format properties. The example below shows the Format tab for an integer text primitive…



As can be seen, the properties shown are indeed identical to those shown on the Format tab of an integer tag. As mentioned above, the properties for the other types of primitive are similarly identical to those of the corresponding tag. You are thus referred to the earlier section of the manual regarding Data Tags for more information on each property.

EDITING THE UNDERLYING TAG

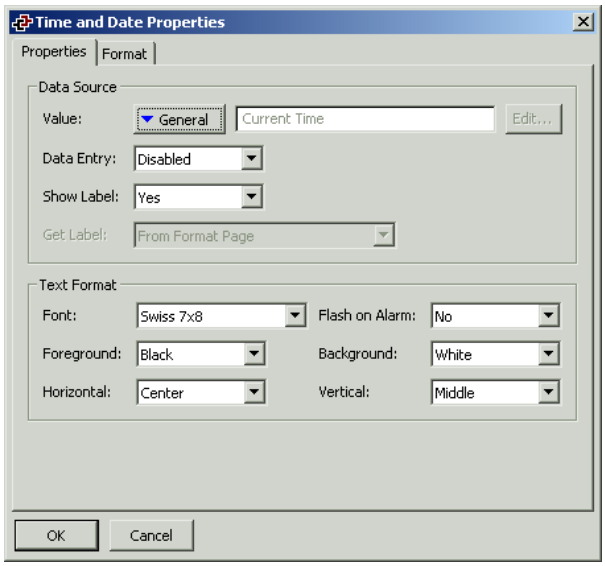
If you want to edit a tag text primitive's properties, either double-click on the primitive, or right-click and select the Properties command from the resulting menu. If, however, you want to edit the properties of the tag that is being used to control the primitive, right-click and select the Tag Details command instead. The resulting dialog box will show the Data and Format tab from the Data Tags window, and allow you to change the various properties. Note that a change made via this mechanism will change all the primitives controlled by that tag if those primitives have the Get Label or Get Format properties set to From Tag Properties.

THE TIME AND DATE PRIMITIVE

The *Time and Date* primitive is used to display the current time and date, or to display the contents of a time and date expression. It can also be used to edit such an expression, or to set the operator panel's real time clock.

The properties of a time and date primitive are displayed using two tabbed pages.

The first page is shown below…

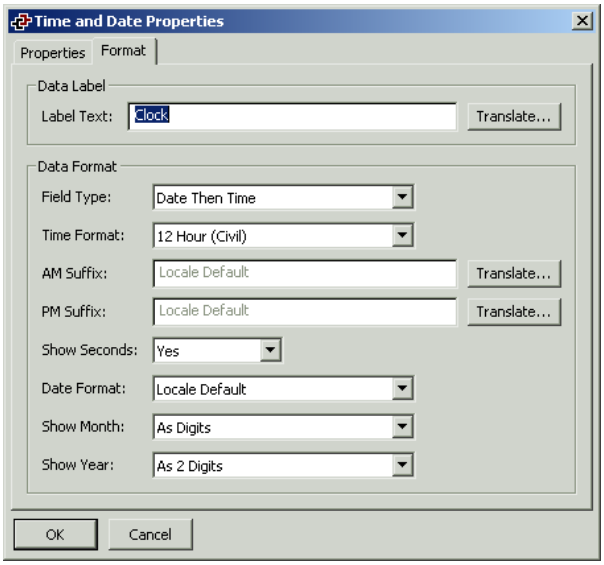


- The *Value* property is used to indicate the time and date value to be displayed. If no value is entered, the current time and date is shown. If an expression is entered, it is taken to represent the number of seconds that have elapsed since 1st January 1997. Such values are typically obtained using the various time and date functions described in the Function Reference.

- The *Data Entry* property is used to indicate whether or not you want the user of the operator interface panel to be able to change the underlying value via this primitive. If no value property has been defined, this amounts to changing the current time or date. If a value property has been entered, the expression entered must be capable of being changed. For example, if a formula is entered, data entry will not be permitted.

- The balance of the properties are as described for tag text primitives. (While it may look odd to have Get Label and Flash On Alarm properties, remember that the value property may be a tag, and so Crimson does have access to the tag label and to the tag's alarm state, should you decide to use them.)

The second tab is shown below…



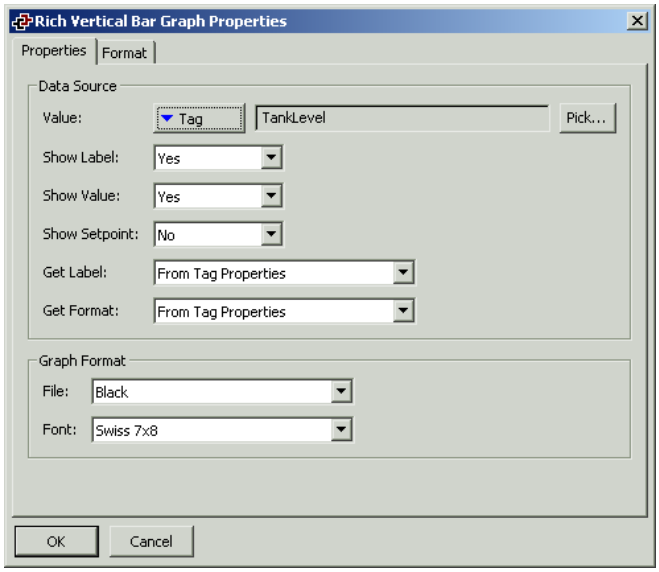
- The *Label Text* property is used to define an optional label for the primitive.
- The *Field Type* property is used to indicate whether the field should display the time, the date or both. In the last case, this property also indicates in which order the two elements should be shown.
- The *Time Format* property is used to indicate whether 12-hour (civil) or 24-hour (military) time format should be used. As with other properties, leaving this set to Locale Default will allow Crimson to pick a suitable format according to the language selected within the operator panel.
- The *AM Suffix* and *PM Suffix* properties are used with 12-hour mode to indicate the text to be appended to the time field in the morning and afternoon as appropriate. If you leave the property undefined, Crimson will use a default.
- The *Show Seconds* property is used to indicate whether the time field should include the seconds, or whether it should just comprise hours and minutes.
- The *Date Format* property is used to indicate the order in which the various date elements (ie. date, month and year) should be displayed.
- The *Show Month* property is used to indicate whether the month should be displayed as digits (ie. 01 through 12) or as its short name (ie. Jan though Dec).
- The *Show Year* property is used to indicate whether the date field should include the year, and if so, how many digits should be shown for that element.

THE RICH BAR-GRAPH PRIMITIVES

The *Rich Vertical Bar* primitive allows you to display a more complex bar-graph which includes a label, a numeric version of the data being displayed, and tick markers to indicate any associated setpoint.

The *Rich Horizontal Bar* primitive allows you to display a more complex bar-graph which includes a label, a numeric version of the data being displayed, and tick markers to indicate any associated setpoint.

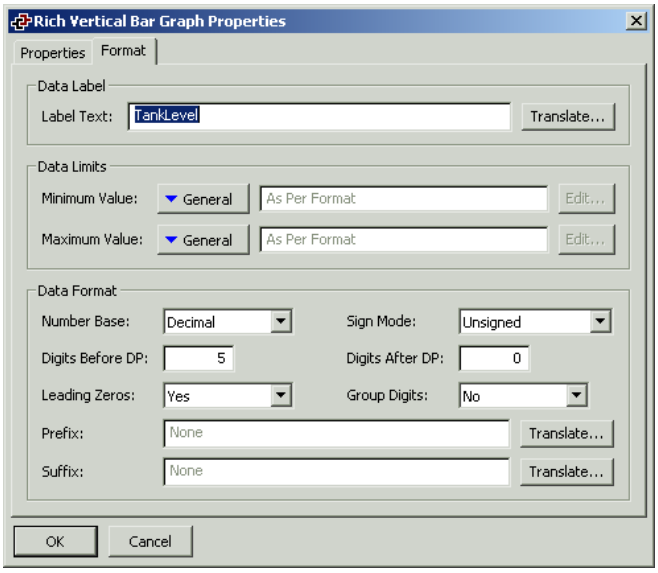
The operation of these rich primitives is analogous to that of the various tag text primitives, in that they are capable of deriving much of the required formatting information from the tag used as their controlling value. Just as with tag text primitives, two tabbed pages are used to edit the primitives' properties. The first of these pages is shown below…



- The *Value* property is used to define the value to be displayed.
- The *Show Label* property is used to indicate whether a label should be included with the bar-graph. For vertical graphs, the label is included at the bottom; for horizontal graphs, it is included at the left-hand side. If a tag is used for the value property, the label may be obtained from that tag. Otherwise, it must be entered on the Format tab of the dialog box.
- The *Show Value* property is used to indicate whether the value of the data should be displayed within the graph itself. If a tag of the appropriate data type is used for the value property, the format may be obtained from the tag. Otherwise, as with the label, it must be entered on the Format tab.
- The *Show Setpoint* property is used to indicate whether tick marks should be added either side of the bar to indicate the setpoint for the controlling value. This option is only available if a tag has been entered for the value field.
- The *Get Label* and *Get Format* properties are as defined for the various tag text primitives. The format is not required if the show value property is set to No.

- The *Fill* property is used to indicate the pattern to be used for the active portion of the bar. If you find that your bar-graph does not appear to work, make sure you have not left this property set to None!
- The *Font* property is used to indicate the font to be used to display the value embedded in the graph, if such a value is enabled via the Show Value property.

The second page contains the label and formatting information for the field...



The properties shown are as described for an integer tag, and you are thus referred to the earlier section of the manual that refers to Data Tags for more information. Note that the existence of this primitive explains why one must enter minimum and maximum values for formulae, when such tags can never be the subject of data entry. If such limits were not defined, how would Crimson know how to scale the bar?

THE SYSTEM PRIMITIVES

The *Alarm Viewer* primitive is used to provide the operator with a method to view and accept active alarms. It will always take up the whole of the display width, but can be restricted to less than the full height if required.

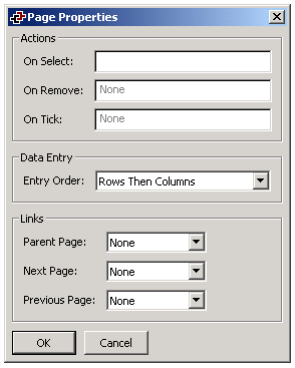
The *Alarm Ticker* primitive scrolls through the active alarms in the system. It takes up a single line, and the whole of the display width. It does not allow the operator to accept the alarms.

The *Event Viewer* primitive is used to provide the operator with a method to view the events recorded in the system's event log. It will always take up the whole of the display width, but can be restricted to less than the full height if required.

If you use manual-accept alarms in your system, you should provide a page which contains an alarm viewer to make sure the operator can accept these alarms. You may also wish to add the alarm ticker to other pages to make the operator aware of alarms while they are viewing other pages. Similarly, if you use events, you should provide a page which contains an event viewer to allow the operator to see what events have occurred.

DEFINING PAGE PROPERTIES

Each page has a number of properties that can be accessed via the Page menu…



- The *On Select* and *On Remove* properties are used to define actions to be performed when the page is first selected for display, or when the page is removed from the display. Refer to the Writing Actions section and the Function Reference for a list of supported actions. Refer to the Data Availability section in this chapter for details of a timeout than can occur when using these properties.
- The *On Tick* property is used to define an action that will run every second during the period for which this page is displayed. Refer to the Writing Actions section and the Function Reference for a list of supported actions. If a lack of data availability results in this action being unable to execute, it will be skipped and retried one second later.
- The *Entry Order* property is used to define how the cursor on the operator panel will move between data entry fields. The settings determine whether fields organized in a grid will be entered in row or column order.
- The *Parent Page* property is used to indicate the page to be displayed when the panel's **`Exit`** key is pressed while this page is active. Selection of this page can be overridden using the techniques below.
- The *Next Page* property is used to indicate the page to be displayed when the panel's **`Next`** key is pressed while this page is active, and when the cursor is on the last data entry field on the page. This selection can also be overridden.
- The *Previous Page* property is used to indicate the page to be displayed when the panel's **`Prev`** key is pressed while this page is active, and when the cursor is on the first data entry field on the page. This selection can also be overridden.

If you have too many data entry fields to fit on a single page, the Next Page and Previous Page properties can be used to link together a series of pages to allow the operator to edit the fields in sequence. Crimson will automatically position the cursor appropriately, such that if the **`Prev`** key is pressed on the first field of a page, the previous page will be activated with the cursor on the last field of that page.

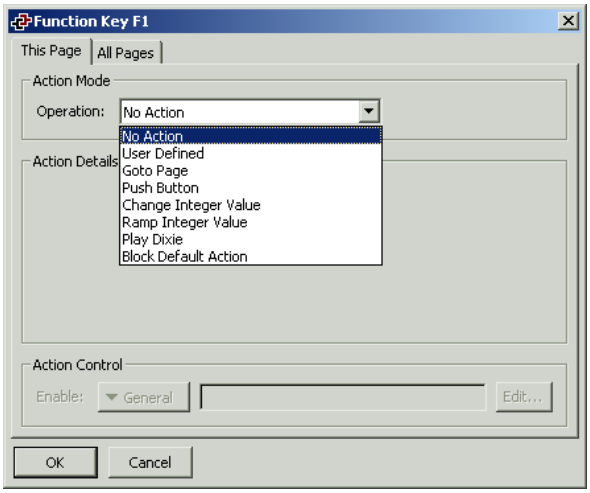
DEFINING SYSTEM ACTIONS

In addition to the various actions that can be defined via page properties, Crimson gives you the ability to define an action to be run when the system first starts, and an action to be run once a second, no matter which page is displayed. These actions can be accessed by selecting the Pages icon in the left-hand pane of the User Interface window.

DEFINING KEY BEHAVIOR

The previous sections have provided a detailed description of how to use the G3's display to get information to the operator. All that remains to complete the User Interface configuration is to define how the operator is to use the G3's keyboard to interact with the system.

To define the actions to be performed by a key, select a zoom level that allows you to see the key in question. For example, if you want to configure one of the function keys, select zoom level one or two as appropriate. Then, double-click the key to display the following…



You will note that this dialog box has two tabbed pages. The first page is used to define what will happen when the key in question is pressed when the current page is selected. The second page is used to define what will happen if the key is pressed when any page is selected. The first type of action is called a *local* action, while the second type is called a *global* action. The color used to display the key will change according to which actions are defined…

If the key is displayed in PURPLE, a local action is defined for this PAGE.

If the key is displayed in GREEN, a GLOBAL action is defined.

If the key is displayed in BLUE, local and global actions are BOTH defined.

Once you have defined an action, you can right-click on the key and use the resulting menu to select either Make Global or Make Local to change the action type. These options will not be available if both types of action have already been defined.

ENABLING ACTIONS

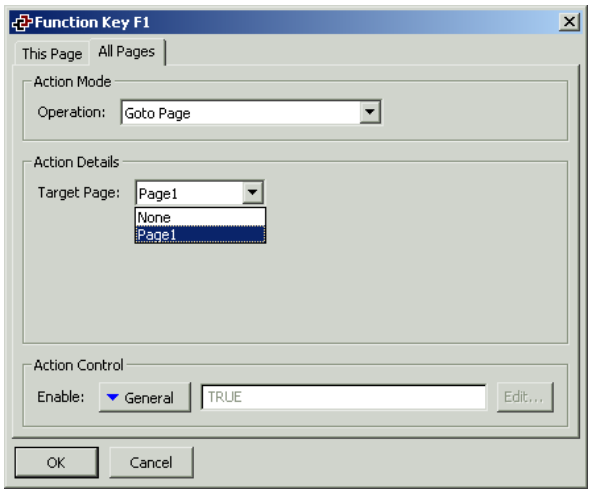
If you want to make a particular action dependent on some condition being true, enter an expression for that condition in the Enable field for the action in question. This expression may reference a flag tag directly, or may use any of the comparison or logical operators defined in the Writing Expressions section. If you need more complex logic such that one of several actions is performed based on more complex decision-making, configure the key in user defined mode and use it to invoke a program that implements the required logic.

ACTION DESCRIPTIONS

The sections below describe each available type of action. When each type is selected, the Action Details portion of the action dialog box will change to show the available options.

THE GOTO PAGE ACTION

This action is used to instruct the G3 to show a new page. The options are shown below…

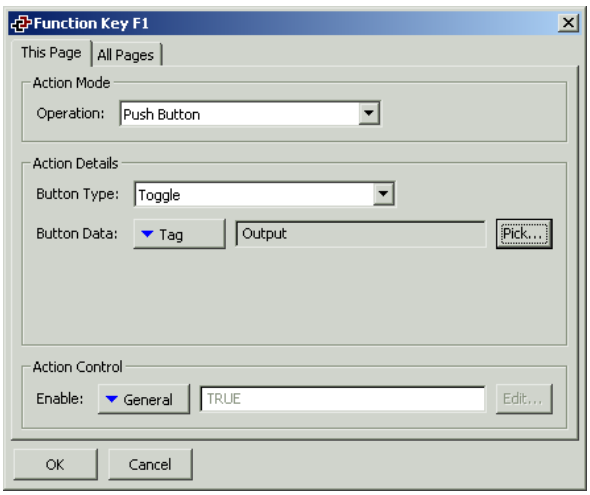


- The *Target Page* property is used to indicate which page should be displayed.

In the example above, the key will select **`Page1`** for display.

THE PUSH BUTTON ACTION

This action is used to emulate a pushbutton. The options are shown below…



- The *Button Type* property is used to define the key's behavior.

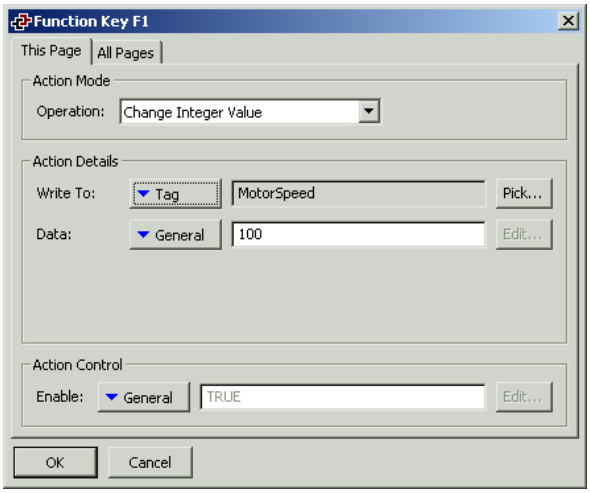
BUTTON TYPE	THE BUTTON WILL…
Toggle	Change the data state when the key is pressed.
Momentary	Set the data to 1 when the key is pressed. Set the data to 0 when the key is released.
Turn On	Set the data to 1 when the key is pressed.
Turn Off	Set the data to 0 when the key is pressed.

- The *Button Data* property is used to define the data to be changed by the key.

In the example above, the key will toggle the value of the `Output` tag.

THE CHANGE INTEGER VALUE ACTION

This action is used to write an integer value to a data item. The options are shown below…

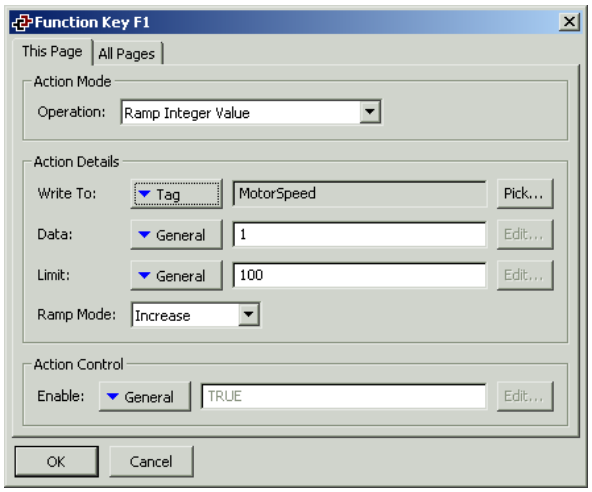


- The *Write To* property is used to define the data item to be changed.
- The *Data* property is used to define the data to be written.

In the example above, the key will set the **`MotorSpeed`** tag to 100.

THE RAMP INTEGER VALUE ACTION

This action is used to increase or decrease a data item. The options are shown below…

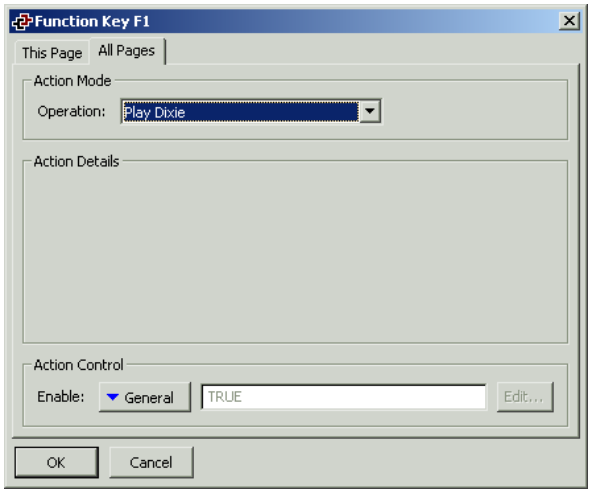


- The *Write To* property is used to define the data item to be changed.
- The *Data* property is used to define the step by which to raise or lower the item.
- The *Limit* property is used to define the minimum or maximum data value.
- The *Ramp Mode* property is used to define whether to raise or lower the item.

In the example above, holding the key will raise **`MotorSpeed`** by 1 until it reaches 100.

THE PLAY DIXIE ACTION

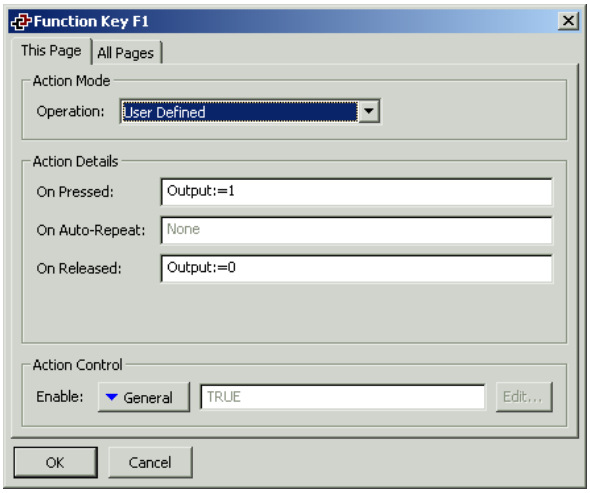
This action plays "Dixie" using the G3's internal sounder. It has no options.



Quite how this action managed to make it into the final release build is anyone's guess, but it is reasonable to assume that later releases will be extended to support playing "Yankee Doodle Dandy" to avoid antagonizing those to the north of the Mason-Dixon line.

THE USER DEFINED ACTION

This action is used to do anything else you desire! The options are shown below…



- The *On Pressed* property is used to define the action to be performed when the key is pressed. This action may invoke any of the functions from the Function Reference or the data modification operators described in the Writing Actions section, or it may run a program.
- The *On Auto-Repeat* property is used to define the action to be performed when the key is pressed and then held down. The action occurs both on the initial depression and on subsequent auto-repeats, so there is no need to define both this property and On Pressed. This action may invoke any of the functions from the Function Reference or the data modification operators described in the Writing Actions section, or it may run a program.
- The *On Released* property is used to define the action to be performed when the key is released. This action may invoke any of the functions from the Function Reference or the data modification operators described in the Writing Actions section, or it may run a program.

In the example above, a user defined action is used to implement a momentary pushbutton.

BLOCK DEFAULT ACTION

This action does not actually do anything, but can be used as a place-holder to prevent further processing. As an example, suppose you have configured **`F1`** to perform a global action, but want to prevent this action from being invoked on a particular page. By configuring **`F1`** on that page as Block Default Action, the global action will not occur.

CHANGING THE LANGUAGE

To configure a key to change the language displayed by the operator panel, select User Defined mode and enter **`SetLanguage(n)`** as the On Pressed property, where **`n`** is a number between 1 and 8, according to the language to be displayed. The display page will be redrawn in the selected language, with any text for which translations have been entered—including fixed text, tag labels and tag formatting information—adjusted as appropriate. Pages that are subsequently displayed will also be drawn in the selected language.

ADVANCED TOPICS

The following sections deal with more advanced issues relating to keyboard actions.

ACTION PROCESSING

When a key is pressed or released, Crimson goes through a defined sequence when deciding what to do with the event. If any stage results in some action being performed, the sequence is stopped, and the later stages do not get a chance to process the key.

The sequence is as follows…

1. If a display primitive is selected for user interaction, it is given a chance to process the key. Active data entry fields will consume the **`Raise`**, **`Lower`**, **`Exit`** and **`Enter`** keys, plus whatever other keys are appropriate to the operation being performed. For example, integer entry fields will also consume the numeric keys.
2. If a display primitive is selected for user interaction and the **`Next`** or **`Prev`** keys are pressed, Crimson will attempt to find the next or previous display primitive which also desires user interaction. If any such field exists, the key will be consumed, and that primitive will be activated.
3. If a local action is defined, the action is performed and the key consumed.
4. If a global action is defined, the action is performed and the key consumed.
5. If the key remains unconsumed, the default actions are implemented…

EVENT	ACTION
`Next` Key Pressed	Displays the page's Next Page, if one is defined.
`Prev` Key Pressed	Displays the page's Previous Page, if one is defined.
`Exit` Key Pressed	Displays the page's Parent Page, if one is defined.
`Menu` Key Pressed	Displays the fist page in the page list.
`Mute` Key Pressed	Silences the G3's internal alarm sounder.

As mentioned above, configuring a key for any global or local action—even one that does nothing, such as Block Default Action—prevents this sequence from proceeding. It should be obvious, then, why such an action is useful, even though at first sight it serves no purpose!

DATA AVAILABILITY

Crimson's communications infrastructure reads only those data items which are required for the current page. This means that when a page is first selected, certain data items may not be available. For a display primitive, this is no problem, as the primitive simply displays an undefined state (typically a number of dashes) until the data becomes available. For actions, though, things can get more complex.

For example, suppose a local action increases the speed of a motor by 50 rpm. If the motor speed is not referenced on the previously displayed page, then, when the page is first displayed, Crimson will not know the current speed, and will thus be unable to write the new value. To handle this, if the operator attempts to perform an action for which the required data is not available, the G3 panel will display a "NOT READY" message until the key in question is released. The operator must then wait a short while, and try the operation again. In practice, communications updates normally take place quickly enough that even the most nimble-fingered operator will be hard pressed to get the message to appear, but since it may on occasions be seen, it is worth explaining.

A slightly more complex issue comes about if the action defined by a page's On Select property is unable to proceed because it also finds that required data is not available. Here, Crimson will wait up to thirty seconds for the data to arrive. If it does not, the action will not be performed, and a "TIMEOUT" message will be displayed for the operator. This timeout mechanism is required to avoid problems should a communications link become severed.

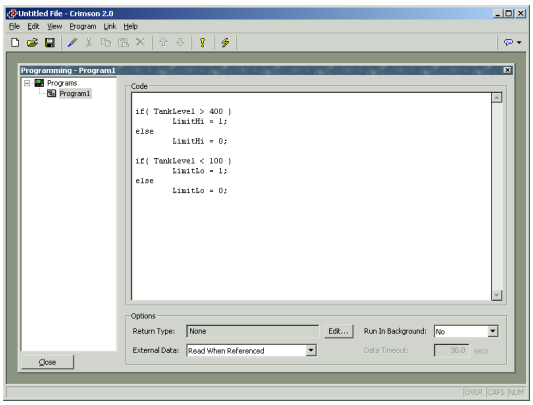
NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

- Pages no longer have text and graphic layers, as all primitives are graphical in nature. This means that the concept of a page format is similarly redundant.
- Page categories have been replaced with system primitives. Where Edict would use an entire page for its alarm viewer, for example, the corresponding system primitive can be used to allocate as little or as much of the display as is required.
- The actions defined by double-clicking a key replace the global and local event maps. If your application used more than one row per event, you will most likely need to use a program to implement the required logic.
- Events such as comms update complete and one second tick have been removed, as most of the actions performed by such events can now be handled via other mechanisms. For example, comms update complete was often used to move data between devices. This can now be performed using the protocol conversion functionality of the Communications window. Also, these events were often misused and lead to the creation of overly complex databases.
- While Edict would typically manage something between two and five display updates per second, Crimson is designed to redraw the display every 100msec, thus providing, for example, smoother operator feedback during data entry.

CONFIGURING PROGRAMS

The previous sections of this manual describe how you can use actions to perform all manner of operations in response to key presses or changes in data tags. If you need to perform an action that is too complex to fit on a single line, or that demands more complex decision-making logic, you can use the Programming icon from the main screen to create and manipulate programs. You should note that many applications will not need programs. You may thus choose to skip this chapter if desired.



USING THE PROGRAM LIST

To create, rename or delete programs, click on the left-hand pane of the User Interface window. The various commands on the Program menu can then be used to make the desired changes. Alternatively, right-click on the required program, and select from the menu.

To select a program, either click on the name in the list, or use the up and down arrows in the toolbar. Alternatively, you can use the **`Alt+Left`** and **`Alt+Right`** key combinations to move up and down the list as required. These keys will work no matter which pane is selected.

EDITING PROGRAMS

To edit a program, simply edit the program text using the large area in the right-hand pane of the Programming window. When you have finished, press the **`Ctrl+T`** key combination or select the Translate command from the Program menu. This will read the program and check it for errors. If an error is found, a dialog box will be displayed, and the cursor will be moved to the approximate position of the error. If no errors exist, a dialog box will be displayed to confirm this fact, and the program will be translated into Crimson's internal format for subsequent execution by the operator panel.

PROGRAM PROPERTIES

The various fields at the bottom of the right-hand pane are used to edit program properties…

- The *Return Type* property is used to indicate whether this program should simply perform a series of actions, or whether it will perform a calculation and return the value of that calculation to the user. Programs that return values are described in more detail below.

- The *Run In Background* property is used to indicate whether Crimson should wait for the program to complete execution before continuing with processing whatever task invoked the program. For example, if this property is set to No, running a program in response to a key being pressed will result in a pause in display updates until the program completes. (Since most programs take very little time to execute, this may not even be noticeable.) If this property is set to Yes, display updates will continue immediately, and the program will execute at a lower priority in the background. Only one background program will run at once, so subsequent requests are queued for later execution. Note also that programs which return values cannot be run in the background, as their return value would then not be available for the caller to use!

- The *External Data* and *Timeout* properties are used to control how the program interacts with Crimson's communication infrastructure with respect to external data items to which the program makes reference. You will recall that Crimson only reads data items when they are used. This property is used to control the exact interpretation of this rule with respect to programs…

MODE	BEHAVIOR
Read When Referenced	External data used by the program will be added to the comms scan whenever the program is referenced. If the program is referenced by a display page, the data will be read when that page is displayed; if the program is referenced by a global action or a trigger, the data will be read at all times. This is the default mode, and is acceptable for all programs, except those that use very large amounts of external data.
Read Always	External data used by the program will be read at all times, whether or not the program is referenced. This means that the program will always be ready to run, and that the operator will not see the "NOT READY" message that might otherwise occur when the program is first referenced. The downside of this mode is that comms performance may be reduced if large amounts of data are referenced by the program.
Read When Executed	External data used within the program will be read only when the program is invoked. The program will wait for the period defined in the timeout property for such data to be available. If the data cannot be read—perhaps because a device is offline—the program will not execute. This mode is typically used with globally-referenced programs that consume large amounts of data that would otherwise slow down the communications scan.
Read But Run Anyway	External data will be treated as described for Read Always mode, but the program will execute whether or not the data has been read successfully. The operator will therefore never see the "NOT READY" message, but if a device is offline, there is no guarantee that the program's data items contain valid data.

ADDING COMMENTS

You can add comments to your programs in two ways. Firstly, you can use the **//** sequence to introduce a comment which will continue for the rest of the current line. Secondly, you can use the **/*** sequence to introduce a single- or multi-line comment. This comment will continue until the ***/** sequence appears. The sample below shows both commenting styles…

```
// This is a single-line comment

/* This is line 1 of the comment
   This is line 2 of the comment
   This is line 3 of the comment */
```

A single-line comment may also be placed at the end of a line which contains code.

RETURNING VALUES

As mentioned above, programs can return values. Such programs can be invoked by other programs or by expressions anywhere in the database. For example, if you want to perform a particularly complex decode on a number of conditions relating to a motor and return a value to indicate the current state, you could create a program that returns an integer like this…

```
if( MotorRunning )
       return 1;
else {
       if( MotorTooHot )
              return 2;
       if( MotorTooCold )
              return 3;
       return 0;
       }
```

You could then configure a multi-state formula to invoke this program, and use that tag's format tab to define the names of the various states. The invocation would be performed by setting the tag's Value property to **Name()**, where **Name** is the name of the program in question. The parentheses are used to indicate a function call, and cannot be omitted.

HERE BE DRAGONS!

Note that you have to exercise a degree of caution when using programs to return values. In particular, you should avoid looping for long periods of time, or performing actions that make no sense in the context in which the function will be invoked. For example, if the code fragment above called the **GotoPage** function to change the page, the display would change every time the program was invoked. Imagine what would happen if you, say, tried to log data from the associated tag, and you'll realize that this would not be a good thing! Therefore, keep programs that return values simple, and always consider the context in which they will be run. If in doubt, avoid doing anything other than simple math and **if** statements.

PROGRAMMING TIPS

The sections below provide an overview of the programming constructions supported by Crimson. The basic syntax used is that of the C programming language. Note that the aim is not to try and teach you to become a programmer, or to master the subtleties of the C language. Such topics are beyond the scope of this manual. Rather, the aim is to provide a quick overview of the facilities available, so that the interested user might experiment further.

MULTIPLE ACTIONS

The simplest type of program comprises a list of actions, with each action taking up a single line, and being followed by a semicolon. All of the various actions defined in the Writing Actions section are available for use. Simple programs like this are typically used where combining the actions in a single action definition would otherwise prove unreadable.

The sample shown below sets several variables, and then changes the display page…

```
Motor1 := 0;
Motor2 := 1;
Motor3 := 0;

GotoPage(Page1);
```

The actions will be executed in order, and the program will then return to the caller.

IF STATEMENTS

This type of statement is used within a program to make a decision. The construct consists of an **`if`** statement with a condition in parentheses, followed by an action (or actions) to be executed if the condition is true. If more than one action is specified, each should be placed on a separate line, and curly-brackets should be used to group the statements together. An optional **`else`** clause can be used to provide for code to be run if the condition is false.

The example below shows an **`if`** statement with a single action…

```
if( TankFull )
       StartPump := 1;
```

The example below shows an **`if`** statement with two actions…

```
if( TankEmpty ) {
       StartPump := 0;
       OpenValue := 1;
       }
```

The example below shows an **if** statement with an **else** clause…

```
if( MotorHot )
       StartFan := 1;
else
       StartFan := 0;
```

Note that it is very important to remember to place the curly-brackets around groups of actions to be executed in the **if** or **else** portion of the statement. If you omit the brackets, Crimson will most likely misunderstand exactly which actions you want to be dependent upon the **if** condition. Although line breaks are recommend between actions, they are not used to figure out what is and is not included within the conditional statement.

SWITCH STATEMENTS

A **switch** statement is used to compare an integer value against a number of possible constants, and to perform an action based upon which value is matched. The exact syntax supports a number of options beyond those shown in the example below, but for the vast majority of applications, this simple form will be acceptable.

This example below will start a motor selected by the value in the **MotorIndex** tag...

```
switch( MotorIndex ) {

       case 1:
              MotorA := 1;
              break;
       case 2:
       case 3:
              MotorB := 1;
              break;
       case 4:
              MotorC := 1;
              break;
       default:
              MotorD := 1;
              break;
       }
```

A value of 1 will start motor A, a value of 2 or 3 will start motor B, and a value of 4 will start motor C. Any value which is not explicitly listed will start motor D. Things to note about the syntax are the use of curly-brackets around the **case** statements, the use of **break** to end each conditional block, the use of two sequential **case** statements to match more than one value, and the use of the optional **default** statement to indicate an action to perform if none of the specified values is matched by the value in the controlling expression. (If this syntax looks too intimidating, a series of **if** statements can be used instead to produce the same results, but with marginally lower performance, and somewhat less readability.)

LOCAL VARIABLES

Some programs use variables to store intermediate results, or to control one of the various loop constructs described below. Rather than defining a tag to hold these values, you can declare what are known as local variables using the syntax shown below…

```
int     a;          // Declare local integer 'a'
float   b;          // Declare local real    'b'
cstring c;          // Declare local string  'c'
```

Local variables may optionally be initialized when they are declared by following the variable name with **:=** and the value to be assigned. Variables which are not initialized in this manner are set to zero, or an empty string, as appropriate.

Note that local variables are truly local in both scope and lifetime. This means that they cannot be referenced outside the program, and they do not retain their values between function invocations. If a function is called recursively, each invocation has its own variables.

LOOP CONSTRUCTS

The three different loop constructs can be used to perform a given section of code while a certain condition is true. The **while** loop tests its condition before the code is executed, while the **do** loop tests the condition afterwards. The **for** loop is a quicker way of defining a **while** loop, allowing you to combine three common elements into one statement.

You should note that some care is required when using loops within your programs, as you may make a programming error which results in a loop that never terminates. Depending on the situation in which the program is invoked, this may seriously disrupt the terminal's user interface activity, or its communications. Loops which iterate too many times may also cause performance issues for the subsystem that invokes them.

THE WHILE LOOP

This type of loop repeats the action that follows it while the condition in the **while** statement remains true. If the condition is never true, the action will never be executed, and the loop will perform no operation beyond evaluating the controlling condition. If you want more than one action to be included in the loop, be sure to surround the multiple statements in curly-brackets, as with the **if** statement. The example below initializes a pair of local variables, and then uses the first to loop through the contents of an array, totaling the first ten elements, and returning the total value to the caller…

```
int i:=0, t:=0;

while( i < 10 ) {
       t := t + Data[i];
       i := i + 1;
       }

return t;
```

The example below shows the same program, but rewritten in a compressed form. Since the loop statement now controls only a single action, the curly-brackets have been omitted…

```
int i:=0, t:=0;

while( i < 10 )
       t += Data[i++];

return t;
```

THE FOR LOOP

You will notice that the **`while`** loop shown above has four elements…

1. The initialization of the loop control variable.
2. The evaluation of a test to see if the loop should continue.
3. The execution of the action to be performed by the loop.
4. The making of a change to the control variable.

The **`for`** loop allows elements 1, 2 and 4 to be combined within a single statement, such that the action following the statement need only implement element 3. This syntax results in something similar to the FOR-NEXT loop found in BASIC and other such languages. Using this statement, the example given above can be rewritten as…

```
int i, t;

for( i:=t:=0; i<10; i++ )
       t += Data[i];

return t;
```

You will notice that the **`for`** statement contains three distinct elements, each separated by semicolons. The first element is the initialization step, which is performed once when the loop first begins; the next is the condition, which is tested at the start of each loop iteration to see if the loop should continue; the final element is the induction step, which is used to make a change to the control variable to move the loop on to its next iteration. Again, remember that if you want more than one action to be included in the loop, include them in curly-brackets!

THE DO LOOP

This type of loop is similar to the **while** loop, except that the condition is tested at the end of the loop. This means that the loop will always execute at least once. The example below shows the example from above, rewritten to use a **do** loop…

```
int i:=0, t:=0;

do {
       t += Data[i];
       } while( ++i < 10 );

return t;
```

LOOP CONTROL

Two additional statements can be used within loops. The **break** statement can be used to terminate the loop early, while the **continue** statement can be used to skip the balance of the loop body and begin another iteration without executing any further code. To make any sense, these statements must be used with **if** statements to make their execution conditional. The example below shows a loop which terminates early if another program returns true…

```
for( i:=0; i<10; i++ ) {
       if( LoopAbort() )
              break;
       LoopBody();
       }
```

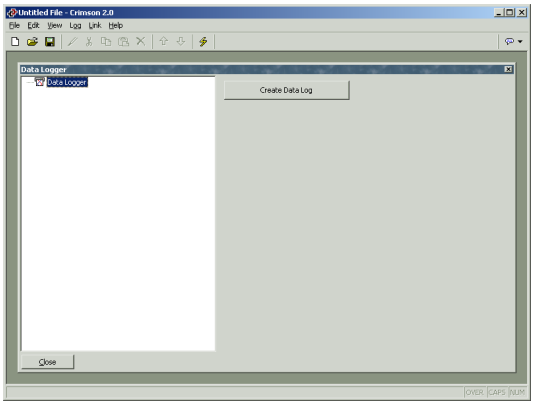
NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

- Crimson supports local variables by means of C-style declarations within the program body, rather than via the local variable table. Unlike Edict's local variables, Crimson's variables are held on the stack, and can thus be used if a program is called recursively. This also means that Crimson's local variables do not hold their values between program invocations.
- Crimson does not support the **Dispatch** function. The decision as to whether to run a program in the foreground or the background is based upon the program's properties, and not on the method used for its invocation.
- Crimson invokes programs using a C-style syntax, and—while the older syntax is still supported—does not need the **Run** function to be used. Programs that return values must be invoked using the newer syntax, though, as the Edict family functions such as **RunInteger** are not provided.
- Programs within Crimson run much more quickly than they did within Edict!

CONFIGURING DATA LOGGING

Now that you have configured the core of your application, you may decide to make use of Crimson's data logger to record certain tag values to CompactFlash. Data recorded in this way is stored in industry-standard comma-separated variable (CSV) files, and can easily be imported into applications such as Excel using a variety of methods. To configure data logging, select the Data Logger icon from the main screen…



CREATING DATA LOGS

You may the use the Create Data Log button to create as many data logs as you need. Since each log can record an unlimited number of data tags, most applications will only use a single log. However, since each log has a fixed set of properties in terms of its sample rate, you may decide to use multiple logs if you wish to sample different data at different rates.

USING THE LOG LIST

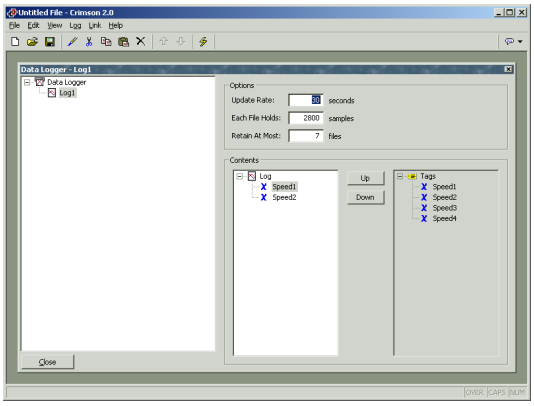
To rename or delete data logs, click on the left-hand pane of the Data Logger window. The commands on the Log menu can then be used to make the desired changes. Alternatively, you may right-click on the required data log, and select from the menu.

(Note that the name of a data log must be eight characters or less in length. This is because the name will be used to define the directory under which the log files are stored, and the G3 panel is not able to handle names that do not conform to FAT-style 8.3 naming.)

To select a data log, either click on the name in the list, or use the up and down arrows in the toolbar. Alternatively, you can use the **`Alt+Left`** and **`Alt+Right`** key combinations to move up and down the list as required. These keys will work no matter which pane is selected.

DATA LOG PROPERTIES

Each data log has the following properties…



- The *Update Rate* property is used to indicate how often Crimson will take a sample of the data items to be logged. The fastest sample rate is one second, but note that using such a high rate will produce very large amounts of data! All of the tags in the log will be sampled at the same rate.

- The *Each File Holds* property is used to indicate how many samples will be included in each log file. When this many samples have been recorded, a new log file will be created using a different name. Typically, this value is set such that each log file contains a sensible amount of data. For example, the log shown above is configured to use a new log file each day.

- The *Retain At Most* property is used to indicate how many log files will be kept on CompactFlash before the oldest file is deleted. This property should be set so as to allow whatever is consuming the logged information to extract the data from the G3 panel before the information is deleted. The log shown above is configured to retain a week's worth of data.

- The *Contents* property is used to indicate which tag should be logged. The first list shows the selected tags, while the second shows those that are available within the database. Tags can be added to the log by double-clicking them in the right-hand list; they can be removed by double-clicking them in the left-hand list, or by pressing the **`Del`** key while the tag is selected. The Up and Down buttons can be used to move tags within the list. One day, someone may even get around to implementing drag-and-drop to allow easier manipulation of this list!

LOG FILE STORAGE

As mentioned above, a data log stores its data in a series of files on the operator panel's CompactFlash card. These files are placed in a subdirectory named after the data log, with this directory being stored under a root directory entry called LOGS. The files are named after the time and date at which the log is scheduled to begin. If each file contains an hour or more of information, the files will be named **`YYMMDDhh.CSV`**, where **`YY`** represents the year of the file, **`MM`** represents the month, **`DD`** represents the date, and **`hh`** represents the hour. If each file

contains less than one hour of information, the files will instead be named **MMDDhhmm.CSV**, with the initial six characters as described above, and the trailing **mm** representing the minute at which the log began. These rules ensure that each log file has a unique name.

THE LOGGING PROCESS

Crimson's data logger operates using two separate processes. The first samples each data point at the rate specified in its properties, and places the logged data into a buffer within the RAM of the G3 panel. The second process executes every two minutes, and writes the data from RAM to the CompactFlash card. This structure has several advantages…

- Writes to the CompactFlash card are guaranteed to begin only on a two-minute boundary—that is, at exactly 2, 4 or 6 minutes past the hour, and so on. This means that if your G3 panel supports hot-swapping of CF cards, you can wait for the next burst of writes to start, and, when the CompactFlash activity LED on the front of the panel ceases to flicker, you are guaranteed to have until the start of the next two-minute interval before further writes will be attempted. This means that you can remove the card without fear of data corruptions. As long as you insert a new card before four minutes have elapsed, no data will be lost.
- Writes to the CompactFlash achieve a much higher level of performance, by avoiding the need to continually update the card's file system data structures for every single sample. For logs configured to sample at very high data rates, the bandwidth of a typical CompactFlash card would not allow data to be written reliably in the absence of such a buffering process.

Note that because data is not committed to CompactFlash for up to two minutes, up to this amount of log data may be lost when the terminal is powered-down. Further, if the terminal is powered-down while a write is in progress, the CompactFlash card may be corrupted. To ensure that such corruption is not permanent, the G3 panel uses a journaling system that caches writes to additional non-volatile memory within the terminal. If the panel detects that a write was interrupted during power-down, the write will be repeated when power is reapplied, thereby reversing any corruption, and repairing the CompactFlash card.

This means that if you want to remove a CompactFlash card from a panel performing data logging, you must observe the procedure described above with respect to the activity LED, and only remove power when the activity has ceased. If you are not sure if the terminal was powered-down correctly, reapply power, allow a CompactFlash write sequence to complete, and power down according to the correct procedure. The card can then be removed safely.

Since the gyrations required to remove a CompactFlash card are somewhat complex, Crimson provides two other mechanisms for accessing log files, thereby eliminating the need for such removals. These methods are described below.

ACCESSING LOG FILES

There are two additional methods of accessing log files…

- The less preferable method is to mount the card as a drive on a PC via the process described at the start of this manual, so that the logs can be copied using Windows Explorer. Note that Windows 2000 or above is recommended when using this method, as earlier versions of Windows may otherwise lock the CompactFlash card and disrupt data logging.
- The preferred method is to use the web server as described in the next chapter. With the web server enabled, log files can be accessed over the panel's Ethernet port, using either a web browser, such as Microsoft Internet Explorer, or by using the automated process implemented by the WebSync utility that is provided with the Crimson configuration software.

USING WEBSYNC

The WebSync utility—which will be stored in the directory specified when the software was installed—can be executed to synchronize a directory on a PC with the contents of an operator panel's data logs. You may decide to configure an application, such as the Windows Scheduler (or perhaps a **cron** daemon), to run this utility on a regular basis, or you may use a command line switch to instruct WebSync to perform the polling automatically. You may also decide to host WebSync on a central server so that the log files can be made available to selected users on your corporate network.

WEBSYNC SYNTAX

WebSync is invoked from the command line using the following syntax…

```
websync {switches} <hostname>
```

…where **<hostname>** is replaced with the IP address of the panel to be polled.

OPTIONAL SWITCHES

The **switches** field may contain one or more of the following options…

- **-terse** can be used to suppress progress information.
- **-poll <n>** can be used to poll the terminal every **n** minutes.
- **-path <dir>** can be used to specify **dir** as the directory to hold the log files.

EXAMPLE USAGE

As an example, the following command line…

```
websync –poll 10 –path C:\Logs 192.9.200.52
```

…will read the log files for all data logs on the terminal with the IP address of 192.9.200.52, and will store these logs under subdirectories of the **C:\Logs** directory. WebSync will continue to execute, and will repeat the polling process every ten minutes. The polling interval must obviously be set such that it is much less than the sampling rate times the number of samples in a file times the number of log files to be retained. If this constraint is met, the directory on the PC will accumulate copies of all the log files from the terminal.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

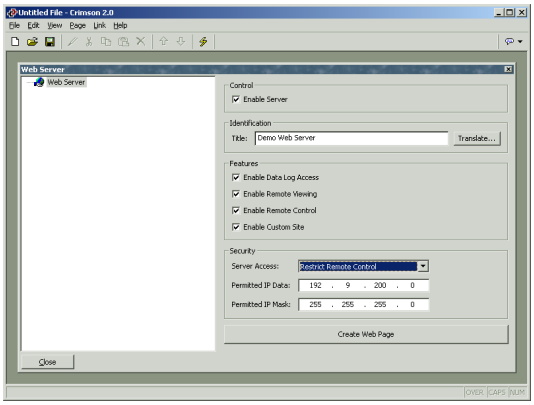
- Logs within Crimson cannot be stopped or started. They operate continuously, creating new log files and removing old ones. The vastly greater storage capacity provided by CompactFlash cards removes the need for logs to be paused.
- Logs within Crimson record the time and date of each sample, and do not need to store the "empty" values used by Edict to mark power-down periods. When a G3 panel is powered-down, this will show simply as a gap in the log files.

CONFIGURING THE WEB SERVER

Crimson's web server can be used to expose various data via the G3 panel's Ethernet port, allowing remote access to diagnostic information, or to the values recorded by the Data Logger. The web server is configured by selecting the Web Server icon from the main screen.

WEB SERVER PROPERTIES

The web server has the following properties…



- The *Enable Server* property is used to enable or disable the web server. If the server is enabled, the panel will monitor port 80 for incoming requests, and will fulfill the requests as required. If the server is disabled, connections to this port will be refused. Remember that in order for the server to operate, the panel's Ethernet port must have been enabled via the Communications window.

- The *Title* property is used to provide the title to be shown on the web server menu. This title can be used to differentiate between several terminals on a network, thereby ensuring that the correct terminal is being accessed.

- The *Enable Data Log Access* property is used to enable or disable web access to the files created by the Data Logger. Obviously, this facility must be enabled if the WebSync utility is to be used to copy the log files to a PC.

- The *Enable Remote Viewing* property is used to enable or disable a facility by which a web browser can be used to view the current contents of a G303's display. This facility is very useful when remotely diagnosing problems that an operator may be having with the operator panel or the machine it controls.

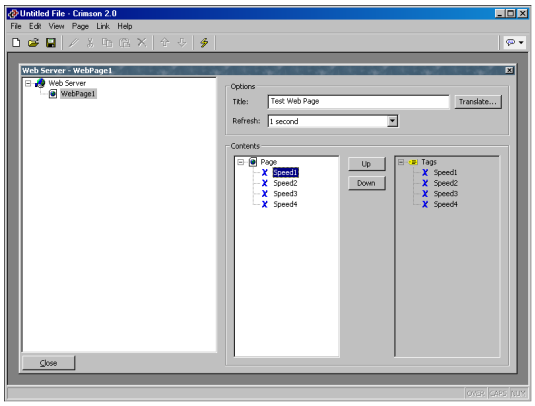
- The *Enable Remote Control* property is used to enable or disable an option by which the remote viewing facility is extended to allow a web browser to be used to simulate the pressing of keys on the operator panel, thereby allowing remote control of the panel or the machine it controls. While this feature is extremely useful, care must be taken to use the various security parameters to avoid unauthorized tampering with a machine. The use of an external firewall is also strongly recommended if the panel is reachable from the Internet.

- The *Enable Custom Site* property is used to enable or disable a facility by which files stored in the WEB directory of the CompactFlash card are exposed via the web server. This facility is described in more detail below.

- The *Security* properties are used to restrict web server access to hosts whose IP address matches the mask and data indicated. All access may be restricted, or the filter may be used to restrict only attempts to use the remote control facility. It is your responsibility to use an external firewall to prevent unauthorized access if the remote control facility is enabled, as the IP filter may be defeated by certain advanced hacking techniques, and is not warranted by Red Lion Controls.

ADDING WEB PAGES

In addition to the facilities described above, the web server supports the display of generic web pages, each of which contains a predefined list of tag values. These pages are created by pressing the Create Web Page button below the web server properties, and are stored in a list similar to that used for display pages, data logs and so on.



Each web page has the following properties…

- The *Title* property is used to identify the web page in the menu presented to the user via their web browser. Although the title is translatable, current versions of Crimson use only the US version of the text.

- The *Refresh* property is used to indicate whether or not the web browser should be instructed to refresh the page contents automatically. Update rates between 1 and 8 seconds are supported. Note that the amount of flicker exhibited by the web browser will vary according to the exact package used and the performance of the machine being employed. The update is not intended to be flicker-free.

- The *Contents* property is used to indicate which tags should be included on the page. The first list shows the selected tags, while the second shows those that are available within the database. Tags can be added to the page by double-clicking them in the right-hand list; they can be removed by double-clicking them in the left-hand list, or by pressing the **`Del`** key while the tag is selected. The Up and Down buttons can be used to move tags within the list. Drag-and-drop operation may one day be implemented to allow easier manipulation of this list!

USING A CUSTOM WEB SITE

While the standard web pages provide quick-and-easy access to the data within the terminal, you may find that your inability to edit their precise formatting leaves your artistic capabilities somewhat frustrated. You may thus use the terminal's custom site facility to create a completely bespoke web site using your favorite third-party HTML editor, and—by inserting certain special sequences and storing the resulting files on the panel's CompactFlash card—expose this site using the panel's web server.

CREATING THE SITE

The web site may use any HTML facilities supported by your browser, but must not use ASP, CGI or other server-side tricks. The filenames used for the HTML files and associated graphics must also comply with the old-style 8.3 naming convention. This means that file extensions will be—for example—**`HTM`** instead or **`HTML`**, and **`JPG`** instead of **`JPEG`**. This also means that the body of the filename must be eight characters or less, and that you must not rely on the difference between upper- and lower-case to differentiate between pages. You may use any directory structure, as long as you once again ensure that your directories observe the 8.3 naming convention and do not rely on case differences.

EMBEDDING DATA

To embed tag data within a web page, insert the sequence **`[[N]]`**, replacing **`N`** with the index number of the tag in question. This index number is displayed on the status bar when a tag is selected within the Data Tag window, and more-or-less corresponds to the order in which the tags were created. When the web page containing this sequence is served, the sequence will be replaced by the current value of the tag, formatted according to the tag's properties.

DEPLOYING THE SITE

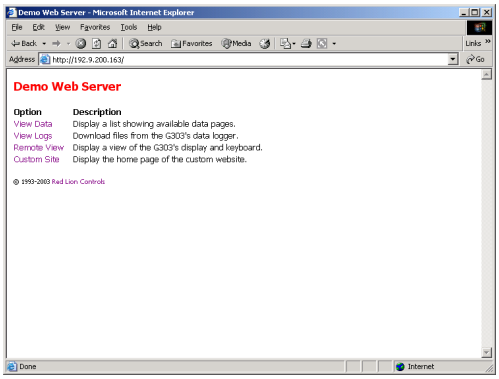
To deploy your custom web site, copy it into the **`\WEB`** directory on the CompactFlash card to be installed in the terminal. To copy the files, either mount the card as a drive on your PC as described at the start of this manual, or use a suitable card writer connected to your PC. Make sure that the *Enable Custom Site* property is set, and the custom site will appear on the web server menu. When the site is selected, a file called **`DEFAULT.HTM`** within the **`\WEB`** directory will be displayed. Beyond that point, navigation is according to the links within the site.

COMPACTFLASH ACCESS

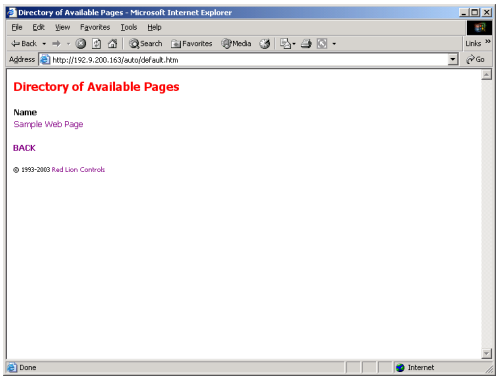
Note that in order to serve custom web pages—or to provide access to the panel's data logger—the web server needs to be able to access the unit's CompactFlash card. If you have mounted the card as a drive on your PC and performed write operations, you may have to wait a minute or so for the PC to unlock the card and allow the terminal to get access. If you are using an operating system earlier than Windows 2000 to perform such an operation, you may find that your PC locks the card when the drive is first mounted, whether or not a write is performed. Again, this lock will be released within a minute or so.

WEB SERVER SAMPLES

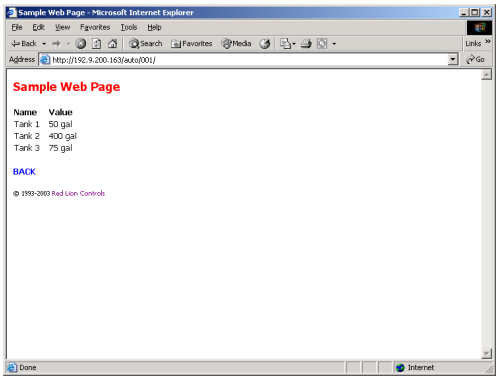
The picture below shows the main menu displayed by the web server…


Demo Web Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links
Address http://192.9.200.163/
Go
Demo Web Server
Option Description
View Data Display a list showing available data pages.
View Logs Download files from the G303's data logger.
Remote View Display a view of the G303's display and keyboard.
Custom Site Display the home page of the custom website.
© 1993-2003 Red Lion Controls
Done
Internet

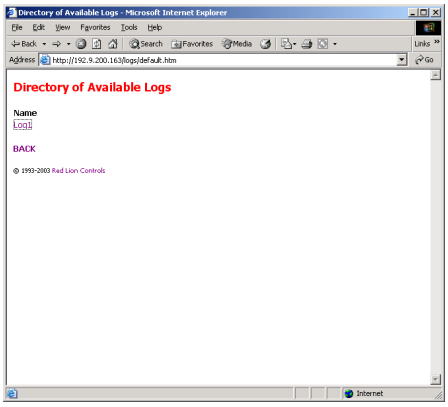
The picture below shows a list of standard web pages…



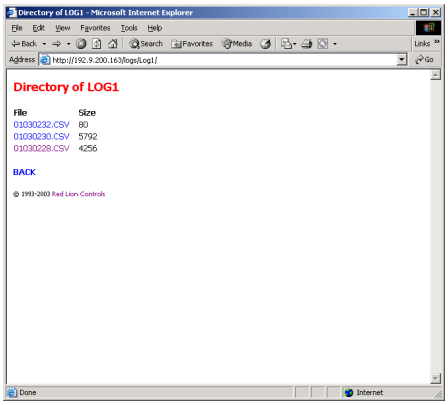
The picture below shows a standard web page containing three tags…



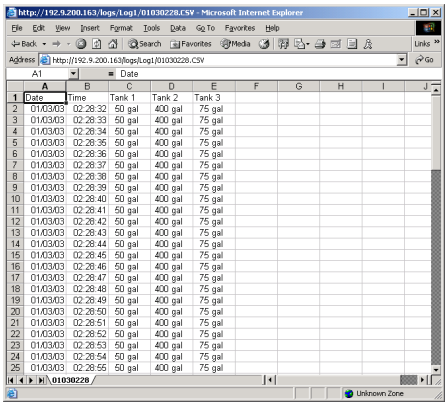
The picture below shows the data log menu…



The picture below shows the contents of a given data log…

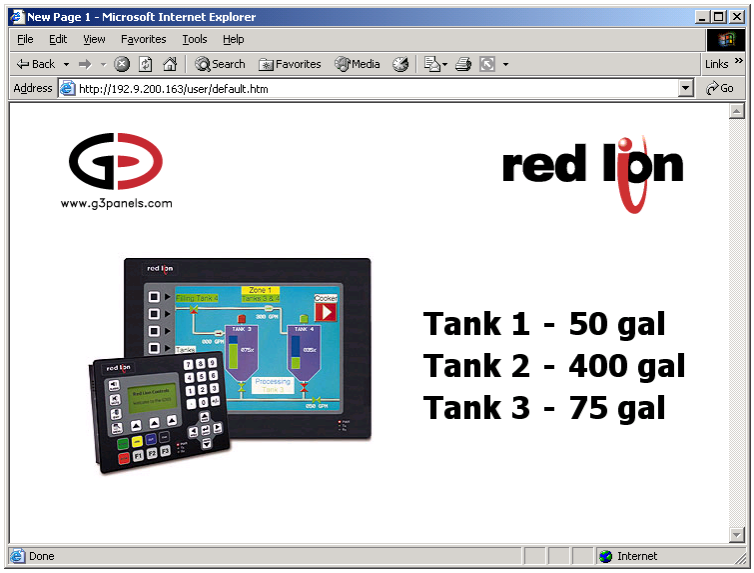


The picture below shows the contents of a given log file…

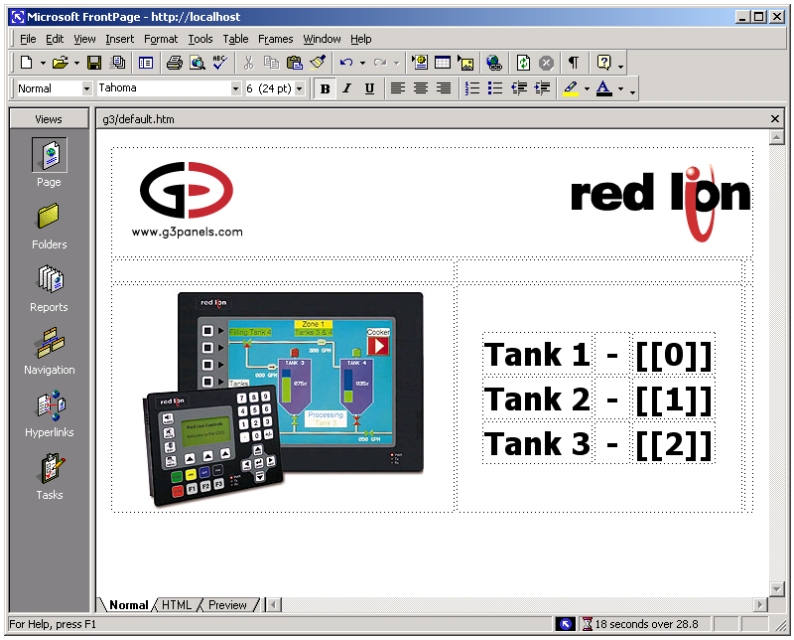


	A	B	C	D	E
1	Date	Time	Tank 1	Tank 2	Tank 3
2	01/03/03	02:28:32	50 gal	400 gal	75 gal
3	01/03/03	02:28:33	50 gal	400 gal	75 gal
4	01/03/03	02:28:34	50 gal	400 gal	75 gal
5	01/03/03	02:28:35	50 gal	400 gal	75 gal
6	01/03/03	02:28:36	50 gal	400 gal	75 gal
7	01/03/03	02:28:37	50 gal	400 gal	75 gal
8	01/03/03	02:28:38	50 gal	400 gal	75 gal
9	01/03/03	02:28:39	50 gal	400 gal	75 gal
10	01/03/03	02:28:40	50 gal	400 gal	75 gal
11	01/03/03	02:28:41	50 gal	400 gal	75 gal
12	01/03/03	02:28:42	50 gal	400 gal	75 gal
13	01/03/03	02:28:43	50 gal	400 gal	75 gal
14	01/03/03	02:28:44	50 gal	400 gal	75 gal
15	01/03/03	02:28:45	50 gal	400 gal	75 gal
16	01/03/03	02:28:46	50 gal	400 gal	75 gal
17	01/03/03	02:28:47	50 gal	400 gal	75 gal
18	01/03/03	02:28:48	50 gal	400 gal	75 gal
19	01/03/03	02:28:49	50 gal	400 gal	75 gal
20	01/03/03	02:28:50	50 gal	400 gal	75 gal
21	01/03/03	02:28:51	50 gal	400 gal	75 gal
22	01/03/03	02:28:52	50 gal	400 gal	75 gal
23	01/03/03	02:28:53	50 gal	400 gal	75 gal
24	01/03/03	02:28:54	50 gal	400 gal	75 gal
25	01/03/03	02:28:55	50 gal	400 gal	75 gal

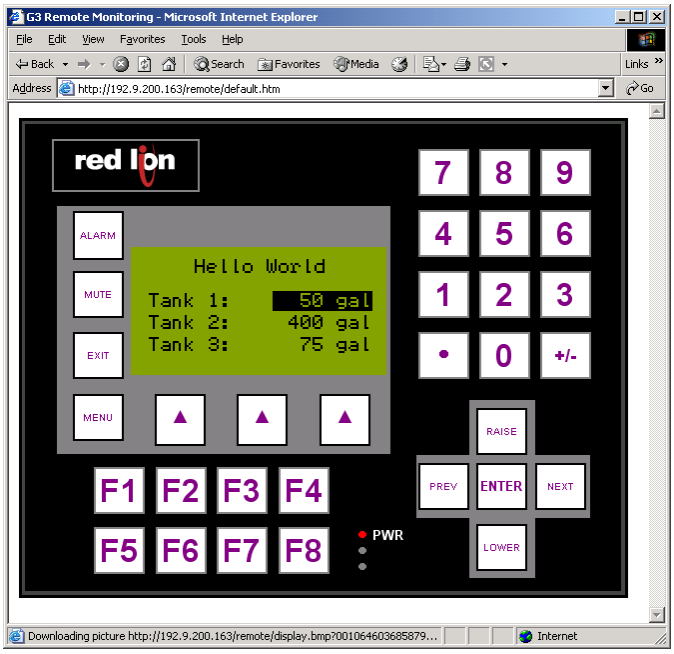
The picture below shows a custom page containing three tags…



The picture below shows the custom page being created within FrontPage…

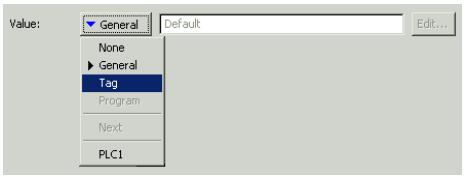


The picture below shows the remote viewing and/or control display…



WRITING EXPRESSIONS

You will recall from the earlier sections of this manual that many fields within Crimson are configured as what are called expression properties. You will further recall that these fields are configured by means of a user interface element similar to that shown below…



In many situations, you will be configuring these properties to be equal to the value of a tag, or to the contents of a register in a remote communications device, in which case your selection will be made simply by clicking the appropriate option on the drop-down menu, and then selecting the required item from the resulting dialog box.

There will be situations, though, when you want to make a property dependent on a more complex combination of data items, perhaps using some math to combine or compare their values. Such eventualities are handled via what are known as expressions, which can be entered in the property's edit box whenever General mode is selected via the drop-down.

DATA VALUES

All expressions contain at least one data value. The simplest expressions are thus references to single constants, single tags or single PLC registers. If you enter either of the last two options, Crimson will simplify the editing process by automatically changing the property mode as appropriate. For example, if you enter a tag name in General mode, Crimson will switch to Tag mode, and show the tag name in the selection field.

CONSTANTS

Constants represent—not surprisingly—constant numbers or strings.

INTEGER CONSTANTS

Integer constants represent a single 32-bit signed number. They may be entered in decimal, binary, octal or hexadecimal as required. The examples below show the same number entered in the four different number bases…

BASE	EXAMPLE
Decimal	`123`
Binary	`0b1111011`
Octal	`0173`
Hexadecimal	`0x7B`

The 'U' and 'L' suffixes supported by earlier versions of software are not used.

CHARACTER CONSTANTS

Character constants represent a single ASCII character, encoded in the lower 8 bits of a 32-bit signed number. A character constant comprises a single character enclosed in single quotation marks, such that **'A'** can be used to represent a value of 65. Certain otherwise unprintable or unrepresentable characters can be encoded using what are called escape sequences, each of which is introduced with a single backslash…

SEQUENCE	VALUE	ASCII
\a	Hex 0x07, Decimal 7	BEL
\t	Hex 0x09, Decimal 9	TAB
\n	Hex 0x0A, Decimal 10	LF
\f	Hex 0x0C, Decimal 12	FF
\r	Hex 0x0D, Decimal 13	CR
\e	Hex 0x1B, Decimal 27	ESC
\x***nnn***	The hex value represented by ***nnn***.	-
nnn	The octal value represented by ***nnn***.	-
\\	A single backslash character.	-
\'	A single quotation mark character.	-
\"	A double quotation mark character.	-

LOGICAL CONSTANTS

Logical constants represent a 1 or 0 value that is used to indicate the truth or otherwise of a yes-or-no expression. An example of something that can be assigned to be equal to a logical constant is a tag that represents a digital output in a PLC. Logical constants can either be entered simply as **1** or **0**, or by use of the keywords **true** or **false**.

FLOATING-POINT CONSTANTS

Floating-point constants represent a 32-bit single-precision floating-point value. They are represented as you might expect—by the integer portion, followed by a single decimal point, followed by the fractional portion. Exponential notation is not supported.

STRING CONSTANTS

String constants represent sequences of characters. They comprise the characters to be represented, enclosed in double quotation marks. For example, the string **"ABCD"** represents a four-character string, comprising the values 65, 66, 67 and 68. (Actually, five bytes are used to store the string, with a null value being appended to indicate the end of the string.) The various escape sequences discussed above may also be used within strings.

TAG VALUES

The value of a tag is represented in an expression by the tag name. Upper-case and lower-case characters are considered equivalent when finding the required tag. Also, once an expression

has been entered, any changes to the name of the tag will modify all of the expressions that make reference to it, so there is no need to re-edit the expressions to "fix" the name.

COMMUNICATIONS REFERENCES

References to registers in master communications devices can be entered into an expression by means of a syntax comprising an opening square bracket, the register name, and a closing square bracket. An optional device name may be prefixed to the register name and separated by a period. The device name need not be specified for registers in the first (or only) device within the database. Examples of this syntax are shown below…

EXAMPLE	MEANING
`[D100]`	Register `D100` in first device.
`[AB.N7:0]`	Register `N7:0` in device `AB`.
`[FX.D100]`	Register `D100` in device `FX`.

SIMPLE MATH

As mentioned above, expressions often contain more than one data value, with their values being combined mathematically. The simplest of these expressions may add a pair of values, while a more complex expression might obtain the average of three values. These operations are performed using the familiar syntax you will have seen in applications such as Excel. The examples below show the basic operations that can be performed…

OPERATOR	PRIORITY	EXAMPLE
Addition	Group 4	`Tag1 + Tag2`
Subtraction	Group 4	`Tag1 - Tag2`
Multiplication	Group 3	`Tag1 * Tag2`
Division	Group 3	`Tag1 / Tag2`
Remainder	Group 3	`Tag1 % Tag2`

Although the examples show spaces surrounding the operators, these are not required.

OPERATOR PRIORITY

You will have noticed the Priority column in the above table. As you no doubt recall from your algebra classes, when several operators are used together, they are evaluated in a defined order. For example, multiplication is always evaluated before addition. Crimson implements this ordering by means of what are known as operator priorities, with each operator being put in a group, and with operators being applied in order from the lowest numbered group to the highest. (Except where noted otherwise in the text, operators within a group are evaluated left-to-right.) The default order of evaluation can be overridden by using parentheses.

TYPE CONVERSION

Normally, Crimson will automatically decide when to switch from evaluating an expression in integer math to evaluating it using floating-point. For example, if you divide an integer

value by a floating-point value, the integer will be converted to floating-point before the division is carried out. However, there will be some situations where you want to force a conversion to take place.

For example, suppose you are adding together three integers which represent the levels in three tanks, and then dividing the total by the tank count to obtain the average level. If you use an expression such as `(Tank1+Tank2+Tank3)/3` then your result may not be as accurate as you demand, as the division will take place using integer math, and the average will not contain any decimal places. To force Crimson to evaluate the result using floating-point math, the simplest technique is to change the `3` to `3.0`, thereby forcing Crimson to convert the sum to floating-point before the division is performed. A slightly more complex technique is to use syntax such as `float(Tank1+Tank2+Tank3)/3`. This invokes what is known as a "type cast" on the term in parentheses, manually converting it to floating-point.

Type casts may also be used to convert a floating-point value to an integer value, perhaps deliberately giving-up some precision from an intermediate value before storing it in a PLC register. For example, the expression `int(cos(Theta)*100)` will calculate the cosine of an angle, multiply this value by 100 using floating-point math, and then convert it to an integer, dropping any digits after the decimal place.

COMPARING VALUES

You will quite often find that you wish to compare the value of one data with another, and make a decision based on the result. For example, you may wish to define a flag formula to show when a tank exceeds a particular value, or you may wish to use an `if` statement in a program to execute some code when a motor reaches its desired speed. The following comparison operators are provided…

OPERATOR	PRIORITY	EXAMPLE
Equal To	Group 7	`Data == 100`
Not Equal To	Group 7	`Data != 100`
Greater Than	Group 6	`Data >  100`
Greater Than or Equal To	Group 6	`Data >= 100`
Less Than	Group 6	`Data <  100`
Less Than or Equal To	Group 6	`Data <= 100`

Each operator produces a value of 0 or 1, depending on the condition it tests. The operators can be used on integers, floating-point values, or text strings. If strings are compared, the comparison is case-insensitive ie. "abc" is considered equal to "ABC".

TESTING BITS

Crimson allows you to test the value of a bit within a data value by using the bit selection operator, which is represented by a single period. The left-hand side of the operator should be the value in which the bit is to be tested, and the right-hand side should be an expression indicating the bit number to test. This right-hand value should be between 0 and 31. The result of the operator is equal to 0 or 1 depending on the value of the bit in question.

OPERATOR	PRIORITY	EXAMPLE
Bit Selection	Group 1	`Input.2`

The example shown tests bit 2 (ie. the bit with a value of 4) within the indicated tag.

If you want to test for a bit being equal to zero, you can use the logical NOT operator…

OPERATOR	PRIORITY	EXAMPLE
Logical NOT	Group 2	`!Input.2`

This example is equal to 1 if bit 2 of the indicated tag is equal to 0, and vice versa.

MULTIPLE CONDITIONS

If you want to define an expression that is true if a number of conditions are *all* true, you can use the logical AND operator. Similarly, if you want to define an expression that is true if *any* of a number of conditions are true, you can use the logical OR operator. The examples below show each operator in use…

OPERATOR	PRIORITY	EXAMPLE
Logical AND	Group 11	`A>10 && B>10`
Logical OR	Group 12	`A>10 || B>10`

The logical AND operator produces a value of 1 if and only if the expressions on the left-hand and right-hand sides are true, while the logical OR operator produces a value of 1 if either expression is true. Note that—unlike the bitwise operators referred to elsewhere in this section—the logical operators stop evaluating once they know what the answer will be. This means that in the above example for logical AND, the right-hand side of the operator will only be evaluated if A is greater than 10, as, if this were not true, the result of the AND operator must already be zero. While this property makes little difference in the examples given above, if the left-hand or right-hand expressions call a program or make a change to a data value, this behavior must be taken into account.

CHOOSING VALUES

You may find situations where you want to select between two values—be they integers, floating-point values or strings—depending on the value of some condition. For example, you may wish to set a motor's speed equal to 500 rpm or 2000 rpm based on a flag tag. This operation can be performed using the **`?:`** operator, which is unique in that it takes three arguments, as shown in the example below…

OPERATOR	PRIORITY	EXAMPLE
Selection	Group 13	`Fast ? 2000 : 500`

This example will evaluate to 2000 if **`Fast`** is true, and 500 otherwise. The operator can be thought to be equivalent to the **`IF`** function found in applications such as Microsoft Excel.

MANIPULATING BITS

Crimson also provides operators to perform operations that do not treat integers as numeric values, but instead as sequences of bits. These operators are known as bitwise operators.

AND, OR AND XOR

These three bitwise operators each produce a result in which each bit is defined to be equal to the corresponding bits in the values on the operator's left-hand and right-hand sides, combined using a specific truth-table…

OPERATOR	PRIORITY	EXAMPLE
Bitwise AND	Group 8	`Data & Mask`
Bitwise OR	Group 9	`Data | Mask`
Bitwise XOR	Group 10	`Data ^ Mask`

The table below shows the associated truth tables…

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

SHIFT OPERATORS

Crimson also provides operators to shift an integer **n** bits to the left or right…

OPERATOR	PRIORITY	EXAMPLE
Shift Left	Group 5	`Data << 2`
Shift Right	Group 5	`Data >> 2`

Each example shifts **Data** two bits in the specified direction.

BITWISE NOT

Finally, Crimson provides a bitwise NOT operator to invert the sense of the bits in a value…

OPERATOR	PRIORITY	EXAMPLE
Bitwise NOT	Group 2	`~Mask`

This example produces a value where every bit is equal to the opposite of its value in **Mask**.

INDEXING ARRAYS

Elements within an array tag can be selected by following the array name with square brackets that contain an indexing expression. This expression must range from 0 to one less

than the number of elements in the array. If you create a 10-element array, for example, the first element will be **Name[0]** and the last will be **Name[9]**.

INDEXING STRINGS

Square brackets can also be used to select characters within a string. For example, if you have a tag called Text that contains the string “ABCD”, then the expression **Text[0]** will return a value of 65, this being equal to the ASCII value of the first character. Index values beyond the end of the string will always return zero.

ADDING STRINGS

As well as adding numbers, the addition operator can be used to concatenate strings. Thus, the expression "AB"+"CD" evaluates to "ABCD”. You may also use the addition operator to add an integer to a string, in which case a single character equal to the ASCII code represented by the integer is appended to the data in the string.

CALLING PROGRAMS

Programs that return values may be invoked within expressions by following the program name with a pair of parentheses. For example, **Program1()*10** will invoke the associated program, and multiply the return value by 10. Obviously, the return type for **Program1** must be set to integer or floating-point for this to make sense.

USING FUNCTIONS

Crimson provides a number of predefined functions that can be used to access system information, or to perform common math operations. These functions are defined in detail in the Function Reference. They are invoked using a syntax similar to that for programs, with any arguments to the function being enclosed within the parentheses. For example, **cos(0)** will invoke the cosine function with an argument of 0, returning a value of +1.0.

PRIORITY SUMMARY

The table below shows the priority of all the operators defined in this section…

GROUP	OPERATORS
Group 1	.
Group 2	! ~
Group 3	* / %
Group 4	+ -
Group 5	<< >>
Group 6	< > <= >=
Group 7	== !=
Group 8	&
Group 9	|
Group 10	^

GROUP	OPERATORS
Group 11	&&
Group 12	||
Group 13	?:

Operators in the lower-numbered groups are applied first.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

- The **&&** and **||** operators stop evaluation once the result is known.
- The only available data types are string, integer and floating-point.
- The **?:** ternary operator is now supported.

WRITING ACTIONS

While expressions are used to define values, actions are used to define what you want to happen when a trigger or other event occurs. Since the vast majority of the actions in a database will relate to key-presses, and since Crimson provides a simple method of defining commonly-used actions via the dialog box discussed in the User Interface section, you will often be able to avoid writing actions "by hand". Actions are needed, though, if you want to use triggers, write programs, or use a key in User Defined mode.

CHANGING PAGE

To create an action that changes the page shown on the panel's display, use the syntax **`GotoPage(Name)`**, where **`Name`** is the name of the display page in question. The current page will be removed, and the new page will be displayed in its place.

CHANGING NUMERIC VALUES

Crimson provides several ways of changing data values.

SIMPLE ASSIGNMENT

To create an action that assigns a new value to a tag or to a register in a communications device, use the syntax **`Data:=Value`**, where **`Data`** is the data item to be changed, and **`Value`** is the value to be assigned. Note that **`Value`** need not just be a constant value, but can be any valid expression of the correct type. Refer to the previous section for details of how to write expressions. For example, code such as **`[N7:0]:=Tank1+Tank2`** can be used to add two tank levels and store the total quantity directly in a PLC register.

COMPOUND ASSIGNMENT

To create an action that sets a data value equal to its current value combined with another value by means of any of the operators defined in the previous section, use the syntax **`Data`***op*`=Value`, where **`Data`** is the tag to be changed, **`Value`** is the value to be used by the operator, and *op* is any of the available operators. For example, the code **`Tag+=10`** will increase **`Tag`** by a value of 10, while **`Tag*=10`** will multiply the current value by 10.

INCREMENT AND DECREMENT

To create an action that increases a data value by one, use the syntax **`Data++`**. To create an action that decreases a tag by one, use the syntax **`Data--`**. Note that the **`++`** or **`--`** operators may be placed before or after the data value in question. In the former case, the value of the expression represented by **`++Data`** is equal to the value of **`Data`** *after* it has been incremented. In the latter case, the expression is equal to the value *before* it has changed.

CHANGING BIT VALUES

To change a bit within a tag, use the syntax **`Data.Bit:=1`** or **`Data.Bit:=0`** to set or clear the bit as required, where **`Data`** is the tag in question and **`Bit`** is the zero-based bit number. Note again that the value on the right-hand side of the **`:=`** operator can be an expression if desired,

such that an example such as **`Data.1:=(Level>10)`** can be used to set or clear a bit depending on whether or not a tank level exceeds a preset value.

RUNNING PROGRAMS

Programs may be invoked within actions by following the program name with a pair of parentheses. For example, **`Program1()`** will invoke the associated program. The program will execute in the foreground or background as defined by the program's properties.

USING FUNCTIONS

Crimson provides a number of predefined functions that can be used to perform various operations. These functions are defined in detail in the Function Reference. They are invoked using a syntax similar to that for programs, with any arguments to the function being enclosed within the parentheses. For example, **`SetLanguage(1)`** will set the terminal language to 1.

OPERATOR PRIORITY

All assignment operators fall into Group 14. In other words, they will be evaluated after all other operators in an action. They are also unique in that they group right-to-left. This means that code such as **`Tag1:=Tag2:=Tag3:=0`** can be used to clear all three tags at once.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

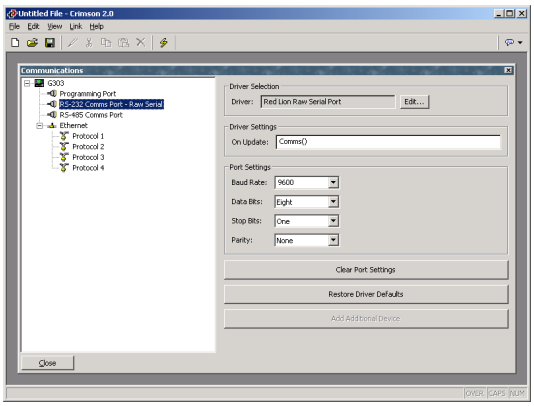
- The = operator can now be used instead of the := operator.

USING RAW PORTS

In order to allow customers to implement simple ASCII protocols without having to ask Red Lion to develop custom drivers, Crimson provides a new facility whereby the software's programming language can be used to directly control either serial ports or TCP/IP network sockets. This functionality—known as raw port access—replaces the driver and functions used to support the Roll-Your-Own Protocol facility within Edict-97. It also replaces the General ASCII Frame protocol by providing a function to perform the parsing operations that the driver previously implemented. Note that if you are not using custom ASCII protocols, but are instead using the standard drivers provided with Crimson, you can skip this section.

CONFIGURING A SERIAL PORT

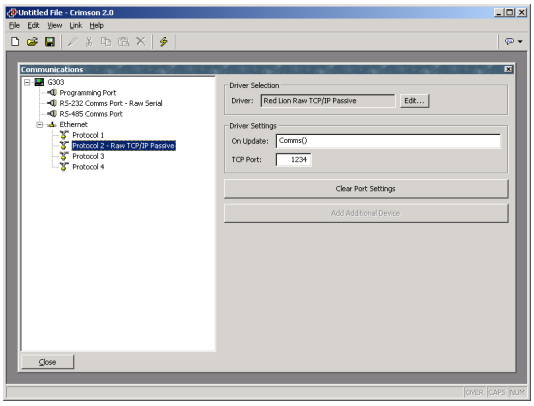
To use a serial port in raw mode, select the Raw Serial Port driver as shown…



The port's Baud rate and other byte format parameters should be configured to indicate the required communications settings, and the On Update property should be set to specify the program that will be performing the communication. This program will be called continually by the port's communications task.

CONFIGURING A TCP/IP SOCKET

To use a TCP/IP socket in raw mode, select the Raw TCP/IP Passive driver as shown…



The On Update property is configured as described above, while the Port property should be configured to indicate which TCP port you want the driver to monitor. The driver will accept connections on this port, and then call the On Update program to handle communications.

READING CHARACTERS

To read data from a raw port a character at a time, use the **PortRead** function, as documented in the Function Reference section of this manual. As with all raw port functions, the ***port*** argument for this function is calculated by counting down the list of ports in the left-hand pane of the Communications window, with the programming port being port 1.

The example below shows to use **PortRead** to accept characters…

```
int Data;

for(;;) {

       if( (Data := PortRead(2, 100)) >= 0 ) {

              /* Add code to process data */
              }
       }
```

Note that by passing a non-zero value for the ***period*** argument, the need to call the **Sleep** function is removed. If you use a zero value for this argument, you must make sure that you suspend the communications task at some point, or you will disrupt system operation.

READING ENTIRE FRAMES

To read an entire frame from a raw port, use the **PortInput** function, as documented in the Function Reference section of this manual. This function allows you to specify frame delimiters, the required frame length and a frame timeout, thereby removing the need to write your own receive state machine. As sample program is shown below…

```
cstring input;
int     value;

for(;;) {

       input := PortInput(5, 42, 13, 0, 0);

       if( value := TextToInt(input, 10) ) {

              Speed := value;

              PortPrint(5, "Value is ");
              PortPrint(5, IntToText(value,10,5));
              PortPrint(5, "\r\n");
              }
       }
```

The example above listens on a TCP/IP socket for a frame which starts with an asterisk and ends with a carriage return. It then converts the frame to a decimal value, stores this in an integer tag, and echoes the value back to the client.

SENDING DATA

To send data on a raw port, use the **`PortWrite`** or **`PortPrint`** functions, as documented in the Function Reference section of this manual. The first function sends a single byte, while the second function sends an entire string. To send numeric values, use the **`IntToText`** function to convert them into strings.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

- The raw serial port device driver controls the port's handshaking lines, so there is no need to call **`SetRTS`**, **`HoldTx`** or any of the various other port management functions. These functions are thus not provided by Crimson.
- When sending data, Crimson automatically handles buffer overflow events, and ensures that no data is lost. The **`PortWrite`** and **`PortPrint`** thus neither provide a return value, nor is such a value required.
- To directly emulate GAF, use a program similar to the example shown above, but store the received string in a string tag, and increment an integer tag. Such direct emulation is not recommended, as you will nearly always then have a trigger to respond to the change in the sequence number, in which case you might as well handle this logic within the communications program.
- Crimson's enhanced programming support allows much higher performance levels when using raw port drivers. Performance typically exceeds that of the equivalent Edict configuration by an order of magnitude or more.

FUNCTION REFERENCE

The following pages describe the various functions that exist within Crimson. These functions can be invoked within actions or expressions as described in the previous two chapters. Functions that are marked as *active* may not be used in expressions that are not allowed to change values eg. in the controlling expression of a display primitive. Functions that are marked as *passive* may be used in any context.

NOTES FOR EDICT USERS

Users of Red Lion's Edict-97 software should note…

- The various **`Port`** functions replace the **`Serial`** RYOP functions.

ABS(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	int / float	The value to be processed.

DESCRIPTION

Returns the absolute value of the argument. In other words, if **value** is a positive value, that value will be returned; if **value** is a negative value, a value of the same magnitude but with the opposite sign will be returned.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int or **float**, depending on the type of the ***value*** argument.

EXAMPLE

```
Error := abs(PV - SP)
```

BEEP(*FREQ, PERIOD*)

ARGUMENT	TYPE	DESCRIPTION
`freq`	`int`	The required frequency in semitones.
`period`	`int`	The required period in milliseconds.

DESCRIPTION

Sounds the terminal's beeper for the indicated period at the indicated pitch. Passing a value of zero for ***`period`*** will turn off the beeper. Beep requests are not queued, so calling the function will immediately override any previous calls. For those of you with a musical bent, the ***`freq`*** argument is calibrated in semitones. On a more serious "note", the Beep function can be a useful debugging aid, as it provides an asynchronous method of signaling the handling of an event, or the execution of a program step.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Beep(60, 100)
```

COPY(*DEST*, *SRC*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
dest	int / float	The first array element to be copied to.
src	int / float	The first array element to be copied from.
count	int	The number of elements to be processed.

DESCRIPTION

Copies ***count*** array elements from ***src*** onwards to ***dest*** onwards.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Copy(Save[0], Work[0], 100)
```

COS(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	float	The angle, in radians, to be processed.

DESCRIPTION

Returns the cosine of the angle ***theta***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
xp := radius*cos(theta)
```

DATE(*Y, M, D*)

ARGUMENT	TYPE	DESCRIPTION
y	int	The year to be encoded, in four-digit form.
m	int	The month to be encoded, from 1 to 12.
d	int	The date to be encoded, from 1 upwards.

DESCRIPTION

Returns a value representing the indicated date as the number of seconds elapsed since the datum point of 1st January 1997. This value can then be used with other time/date functions.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
t := Date(2000,12,31)
```

DEG2RAD(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	float	The angle to be processed.

DESCRIPTION

Returns ***theta*** converted from degrees to radians.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
Load := Weight * cos(Deg2Rad(Angle))
```

FILL(ELEMENT, DATA, COUNT)

ARGUMENT	TYPE	DESCRIPTION
element	int / float	The first array element to be processed.
data	int / float	The data value to be written.
count	int	The number of elements to be processed.

DESCRIPTION

Sets ***count*** array elements from ***element*** onwards to be equal to ***data***.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Fill(List[0], 0, 100)
```

GETDATE (*TIME*) AND FAMILY

ARGUMENT	TYPE	DESCRIPTION
time	int	The time value to be decoded.

DESCRIPTION

Each member of this family of functions returns some component of a time/date value, as previously created by **GetNow**, **Time** or **Date**. The available functions are as follows…

FUNCTION	DESCRIPTION
GetDate	Returns the day-of-month portion of ***time***.
GetDay	Returns the day-of-week portion of ***time***.
GetDays	Returns the number of days in ***time***.
GetHour	Returns the hours portion of ***time***.
GetMin	Returns the minutes portion of ***time***.
GetMonth	Returns the month portion of ***time***.
GetSec	Returns the seconds portion of ***time***.
GetWeek	Returns the week-of-year portion of ***time***.
GetWeeks	Returns the number of weeks in ***time***.
GetWeekYear	Returns the week year when using week numbers.
GetYear	Returns the year portion of ***time***.

Note that **GetDays** and **GetWeeks** are typically used with the difference between two time values to calculate how long has elapsed in terms of days or weeks. Note also that the year returned by ***GetWeekYear*** is not always the same as that returned by ***GetYear***, as the former may return a smaller value if the last week of a year extends beyond year-end.

FUNCTION TYPE

These functions are passive.

RETURN TYPE

int.

EXAMPLE

```
d := GetDate(GetNow() - 12*60*60)
```

GETMONTHDAYS(*Y, M*)

ARGUMENT	TYPE	DESCRIPTION
y	int	The year to be processed, in four-digit form.
m	int	The month to be processed, from 1 to 12.

DESCRIPTION

Returns the number of days in the indicated month, accounting for leap years etc.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
Days := GetMonthDays(2000, 3)
```

GETNOW()

ARGUMENT	TYPE	DESCRIPTION
None		

DESCRIPTION

Returns the current time and date as the number of seconds elapsed since the datum point of 1st January 1997. This value can then be used with other time/date functions.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
t := GetNow()
```

GOTOPAGE(*NAME*)

ARGUMENT	TYPE	DESCRIPTION
name	Display Page	The page to be displayed.

DESCRIPTION

Selects page ***name*** to be shown on the terminal's display.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
GotoPage(Page1)
```

INTTOTEXT(*DATA*, *RADIX*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
data	int	The value to be processed.
radix	int	The number base to be used.
count	int	The number of digits to generate.

DESCRIPTION

Returns the string obtained by formatting ***data*** in base ***radix***, generating ***count*** digits. The value is assumed to be unsigned, so if a signed value is required, use **Sgn** to decide whether to prefix a negative sign, and then use **Abs** to pass the absolute value to **IntToText**.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring.

EXAMPLE

```
PortPrint(1, IntToText(Value, 10, 4))
```

LEFT(*STRING*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
string	cstring	The string to be processed.
count	int	The number of characters to return.

DESCRIPTION

Returns the first ***count*** characters from ***string***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring.

EXAMPLE

```
AreaCode := Left(Phone, 3)
```

LEN(*STRING*)

ARGUMENT	TYPE	DESCRIPTION
string	cstring	The string to be processed.

DESCRIPTION

Returns the number of characters in ***string***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
Size := Len(Input)
```

MAKEINT(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	float	The value to be converted.

DESCRIPTION

Reinterprets the floating-point argument as an integer. This function does not perform a type conversion, but instead takes the bit pattern stored in the argument, and assumes that rather than representing a floating-point value, it actually represents an integer. It can be used to manipulate data from a remote device that might actually have a different data type from that expected by the communications driver.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
n := MakeInt(fp);
```

MAKEFLOAT(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
value	int	The value to be converted.

DESCRIPTION

Reinterprets the integer argument as a floating-point value. This function does not perform a type conversion, but instead takes the bit pattern stored in the argument, and assumes that rather than representing an integer, it actually represents a floating-point value. It can be used to manipulate data from a remote device that might actually have a different data type from that expected by the communications driver.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
fp := MakeInt(n);
```

MAX(*A*, *B*)

ARGUMENT	TYPE	DESCRIPTION
a	int / float	The first value to be compared.
b	int / float	The second value to be compared.

DESCRIPTION

Returns the larger of the two arguments.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int or **float**, depending on the type of the arguments.

EXAMPLE

```
Larger := Max(Tank1, Tank2)
```

MEAN(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
`element`	`int` / `float`	The first array element to be processed.
`count`	`int`	The number of elements to be processed.

DESCRIPTION

Returns the mean of the ***`count`*** array elements from ***`element`*** onwards.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`float`.

EXAMPLE

```
Average := Mean(Data[0], 10)
```

MID(*STRING*, *POS*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
string	cstring	The string to be processed.
pos	int	The position at which to start.
count	int	The number of characters to return.

DESCRIPTION

Returns ***count*** characters from position ***pos*** within ***string***, where 0 is the first position.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring.

EXAMPLE

```
Exchange := Mid(Phone, 3, 3)
```

MIN(*A, B*)

ARGUMENT	TYPE	DESCRIPTION
a	int / float	The first value to be compared.
b	int / float	The second value to be compared.

DESCRIPTION

Returns the smaller of the two arguments.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int or **float**, depending on the type of the arguments.

EXAMPLE

```
Smaller := Min(Tank1, Tank2)
```

MUTESIREN()

ARGUMENT	TYPE	DESCRIPTION
None		

DESCRIPTION

Turns off the operator panel's internal siren.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
MuteSiren()
```

NOP()

ARGUMENT	TYPE	DESCRIPTION
None		

DESCRIPTION

This function does nothing.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Nop ()
```

PI()

ARGUMENT	TYPE	DESCRIPTION
None		

DESCRIPTION

Returns *pi* as a floating-point number.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
Scale = Pi()/180
```

PLAYRTTTL (*TUNE*)

ARGUMENT	TYPE	DESCRIPTION
tune	cstring	The tune to be played in RTTTL representation.

DESCRIPTION

Plays a tune using the terminal's internal beeper. The **tune** argument should contain the tune to be played in RTTTL format—the format used by a number of cell phones for custom ring tones. Sample tunes can be obtained from many sites on the World Wide Web.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PlayRTTTL("TooSexy:d=4,o=5,b=40:16f,16g,16f,16g,16f.,16f,16g,16f,16g,16g#
.,16g#,16g,16g#,16g,16f.,16f,16g,16f,16g,16f.,16f,16g,16f,16g,16f.,16f,16
g,16f,16g,16g#.,16g#,16g,16g#,16g,16f.,16f,16g,16f,16g,32f.")
```

POPDEV(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	int / float	The first array element to be processed.
count	int	The number of elements to be processed.

DESCRIPTION

Returns the standard deviation of the ***count*** array elements from ***element*** onwards, assuming the data points to represent the whole of the population under study. If you need to find the standard deviation of a sample, use the **StdDev** function instead.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
Dev := PopDev(Data[0], 10)
```

PORTINPUT(*PORT*, *START*, *END*, *TIMEOUT*, *LENGTH*)

ARGUMENT	TYPE	DESCRIPTION
port	int	The raw port to be read.
start	int	The start character to match, if any.
end	int	The end character to match, if any.
timeout	int	The inter-character timeout in milliseconds, if any.
length	int	The maximum number of characters to read, if any.

DESCRIPTION

Reads a string of characters from the ***port*** indicated by port, using the various other parameters to control the input process. If ***start*** is non-zero, the process begins by waiting until the character indicated by this parameter is received. If ***start*** is zero, the receive process begins immediately. The process then continues until one of the following conditions has been met…

- ***end*** is non-zero and a character matching ***end*** is received.
- ***timeout*** is non-zero, and that period passes without a character being received.
- ***length*** is non-zero, and that many characters have been received.

The function then returns the characters received, not including the ***start*** or ***end*** byte. This function is used together with Raw Port drivers to implement custom protocols using Crimson's programming language. It replaces the RYOP functionality found in Edict.

FUNCTION TYPE

This function is active.

RETURN TYPE

cstring.

EXAMPLE

```
Frame := PortInput(1, '*', 13, 100, 200)
```

PORTPRINT(*PORT, STRING*)

ARGUMENT	TYPE	DESCRIPTION
port	int	The raw port to be written to.
string	cstring	The text string to be transmitted.

DESCRIPTION

Transmits the text contained in ***string*** to the port indicated by ***port***. The port must be configured to use a raw driver, such as the raw serial port driver, or either of the raw TCP/IP drivers. The data will be transmitted, and the function will return. The port driver will handle handshaking and control of transmitter enable lines as required.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PortPrint(1, "ABCD")
```

PORTREAD(*PORT*, *PERIOD*)

ARGUMENT	TYPE	DESCRIPTION
port	int	The raw port to be read.
period	int	The time to wait in milliseconds.

DESCRIPTION

Attempts to read a character from the port indicated by ***port***. The port must be configured to use a raw driver, such as the raw serial port driver, or either of the raw TCP/IP drivers. If no data is available within the indicated time period, a value of –1 will be returned. Setting ***period*** to zero will result in any queued data being returned, but will prevent Crimson from waiting for data to arrive if none is available.

FUNCTION TYPE

This function is active.

RETURN TYPE

int.

EXAMPLE

```
Data := PortRead(1, 100)
```

PORTWRITE(*PORT, DATA*)

ARGUMENT	TYPE	DESCRIPTION
port	int	The raw port to be written to.
data	int	The byte to be transmitted.

DESCRIPTION

Transmits the byte indicated by ***data*** on the port indicated by ***port***. The port must be configured to use a raw driver, such as the raw serial port driver, or either of the raw TCP/IP drivers. The character will be transmitted, and the function will return. The port driver will handle handshaking and control of transmitter enable lines as required.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
PortWrite(1, 'A')
```

POWER(*VALUE*, *POWER*)

ARGUMENT	TYPE	DESCRIPTION
`value`	`int` / `float`	The value to be processed.
`power`	`int` / `float`	The power to which ***`value`*** is to be raised.

DESCRIPTION

Returns ***`value`*** raised to the ***`power`***-th power.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or **`float`**, depending on the type of the ***`value`*** argument.

EXAMPLE

```
Volume := Power(Length, 3)
```

RAD2DEG(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	float	The angle to be processed.

DESCRIPTION

Returns ***theta*** converted from radians to degrees.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
Right:=Rad2Deg(Pi()/2)
```

RANDOM(*RANGE*)

ARGUMENT	TYPE	DESCRIPTION
range	int	The range of random values to produce.

DESCRIPTION

Returns a pseudo-random value between 0 and ***range*-1**.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
Noise := Random(100)
```

RIGHT(*STRING*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
string	cstring	The string to be processed.
count	int	The number of characters to return.

DESCRIPTION

Returns the last ***count*** characters from ***string***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

cstring.

EXAMPLE

```
Local := Left(Phone, 7)
```

SETLANGUAGE(*CODE*)

ARGUMENT	TYPE	DESCRIPTION
Code	int	The language to be selected.

DESCRIPTION

Set the terminal's current language to that indicated by ***code***.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SetLanguage(1)
```

SGN(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
`value`	`int` / `float`	The value to be processed.

DESCRIPTION

Returns –1 if ***value*** is less than zero, +1 if it is greater than zero, or 0 if it is equal to zero.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or **`float`**, depending on the type of the ***value*** argument.

EXAMPLE

```
State:=Sgn(Level)+1
```

SIN(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	float	The angle, in radians, to be processed.

DESCRIPTION

Returns the sine of the angle ***theta***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
yp := radius*sin(theta)
```

SIRENON()

ARGUMENT	TYPE	DESCRIPTION
None		

DESCRIPTION

Turns on the operator panel's internal siren.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
SireOn()
```

SLEEP(*PERIOD*)

ARGUMENT	TYPE	DESCRIPTION
period	int	The period for which to sleep, in milliseconds.

DESCRIPTION

Sleeps the current task for the indicated number of milliseconds. This function is normally used within programs that run in the background, or that implement custom communications using Raw Port drivers. Calling it in response to triggers or key presses is not recommended.

FUNCTION TYPE

This function is active.

RETURN TYPE

This function does not return a value.

EXAMPLE

```
Sleep(100)
```

SQRT(*VALUE*)

ARGUMENT	TYPE	DESCRIPTION
`value`	`int` / `float`	The value to be processed.

DESCRIPTION

Returns the square root of ***value***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`int` or **`float`**, depending on the type of the ***value*** argument.

EXAMPLE

```
Flow := Const * Sqrt(Input)
```

STDDEV(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
`element`	`int` / `float`	The first array element to be processed.
`count`	`int`	The number of elements to be processed.

DESCRIPTION

Returns the standard deviation of the ***`count`*** array elements from ***`element`*** onwards, assuming the data points to represent a sample of the population under study. If you need to find the standard deviation of the whole population, use the **`PopDev`** function instead.

FUNCTION TYPE

This function is passive.

RETURN TYPE

`float`.

EXAMPLE

```
Dev := StdDev(Data[0], 10)
```

SUM(*ELEMENT*, *COUNT*)

ARGUMENT	TYPE	DESCRIPTION
element	int / float	The first array element to be processed.
count	int	The number of elements to be processed.

DESCRIPTION

Returns the sum of the ***count*** array elements from ***element*** onwards.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int or **float**, depending on the type of the ***value*** argument.

EXAMPLE

```
Total := Sum(Data[0], 10)
```

TAN(*THETA*)

ARGUMENT	TYPE	DESCRIPTION
theta	float	The angle, in radians, to be processed.

DESCRIPTION

Returns the tangent of the angle ***theta***.

FUNCTION TYPE

This function is passive.

RETURN TYPE

float.

EXAMPLE

```
yp := xp * tan(theta)
```

TEXTTOINT(*STRING*, *RADIX*)

ARGUMENT	TYPE	DESCRIPTION
string	cstring	The string to be processed.
radix	int	The number base to be used.

DESCRIPTION

Returns the value of ***string***, treating it as a number of base ***radix***. This function is often used together with **Mid** to extract values from strings received from raw serial ports. It can also be used to convert other string values into integers.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
Data := TextToInt("1234", 10)
```

TIME(*H, M, S*)

ARGUMENT	TYPE	DESCRIPTION
h	int	The hour to be encoded, from 0 to 23.
m	int	The minute to be encoded, from 0 to 59.
s	int	The second to be encoded, from 0 to 59.

DESCRIPTION

Returns a value representing the indicated time as the number of seconds elapsed since midnight. This value can then be used with other time/date functions. It can also be added to the value produced by **Date** to produce a value that references a particular time and date.

FUNCTION TYPE

This function is passive.

RETURN TYPE

int.

EXAMPLE

```
t := Date(2000,12,31) + Time(12,30,0)
```